Rapport de stage

Développement d'une plateforme de tests automatisés : GreenT

Antoine de Roquemaurel

M1 Informatique – Développement Logiciel
2014 – 2015

Maître de stage :
Alain Fernandez

Tuteur universitaire :
Bernard Cherbonneau

Du 04 Mai au 28 Août 2015
Version du 25 août 2015

Je tiens à remercier toutes les personnes m'ayant permis de réaliser ce stage.

En premier lieu, un grand merci à Corinne Tarin pour m'avoir accepté au sein de son équipe.
Je remercie particulièrement Alain Fernandez pour m'avoir suivi et conseillé tout au long de ce stage tout en partageant son expérience.
Une pensée à Joelle Decol pour sa bonne humeur quotidienne, ainsi qu'à toute l'équipe du troisième étage, grâce à qui j'ai passé d'excellents moments au sein de l'entreprise.
Merci à mon tuteur universitaire Bernard Cherbonneau pour son suivi et sa visite en entreprise.
Enfin, je remercie toutes les personnes m'ayant entouré durant ce stage et aidé à la rédaction ce rapport, à savoir Diane, Ophélie, Clément et Mathieu.

Introduction

Dans le cadre de ma formation en première année de Master spécialité Développement Logiciel à l'université Toulouse III – Paul Sabatier, j'ai eu la possibilité d'effectuer un stage d'une durée de quatre mois.

Attiré par le monde de l'entreprise et désireux de gagner en expérience, j'ai pu continuer un projet commencé l'année précédente dans l'entreprise Continental Automotive : le développement d'une plateforme de tests de logiciels embarqués.

Ce projet a pour but d'aider une équipe de Continental. Celle-ci est face à un problème : l'intégration d'un plugin dans un logiciel de contrôle moteur. Celui-ci possède des centaines de variables interfaces et est donc compliqué à tester. Afin d'aider cette équipe, une plateforme permettant d'effectuer des tests automatiques est en développement : *GreenT*.

Ce projet est encore aujourd'hui en cours de développement, et pour pouvoir disposer d'une première version de production, il est nécessaire d'implémenter de nouvelles fonctionnalités et de corriger des bugs. Ayant connu les prémices de ce projet, et afin d'avoir un aperçu de celui-ci sur la durée, allant de sa conception jusqu'à sa mise en production, le sujet du stage était particulièrement intéressant. En outre, celui-ci est en parfaite adéquation avec mon projet professionnel, et ma poursuite en M2 Développement Logiciel. En effet, ce projet est au cœur du problème d'ingénieur logiciel, par la problématique que celui-ci essaye de régler : la manière dont nous pouvons tester un logiciel.

J'ai travaillé au sein de l'équipe *Test & Automation Service*, je vais ainsi vous présenter en quoi le développement de cet outil est nécessaire à l'équipe en charge des tests de ce plugin.
Dans une première partie nous présenterons l'entreprise Continental et plus particulièrement l'équipe *Tests & Automation Service*(chapitre 1), puis nous verrons le problème que posent actuellement les tests de ce plugin(chapitre 2), nous aborderons ensuite la manière dont nous nous sommes organisés pour le développement (chapitre 3) avant de présenter la solution qui est en cours de développement(chapitre 4) et comment j'ai contribué à ce projet(chapitre 5).

Table des matières

1 Continental

Mon stage s'est déroulé au sein de l'entreprise Continental. Cette entreprise est, sur les dernières années, entre première et deuxième équipementier automobile mondial par le volume de vente.

Sommaire

Ce chapitre présente rapidement le groupe Continental et plus particulièrement l'équipe *Tests & Automation Service* qui m'a accueilli pour ce stage.

1.1 Organisation de l'entreprise

1.1.1 Le groupe Continental

Continental est une entreprise allemande fondée en 1871 dont le siège se situe à Hanovre. Il s'agit d'une Société Anonyme (SA) dont le président du comité de direction est le Dr. Elmar Degenhart depuis le 12 août 2009. Le groupe Continental est constitué de cinq divisions intervenant sur le marché des pneus (*Rubber*) et de l'électronique automobile (*Automotive*), ces divisions vous sont détaillés figure 1.4.

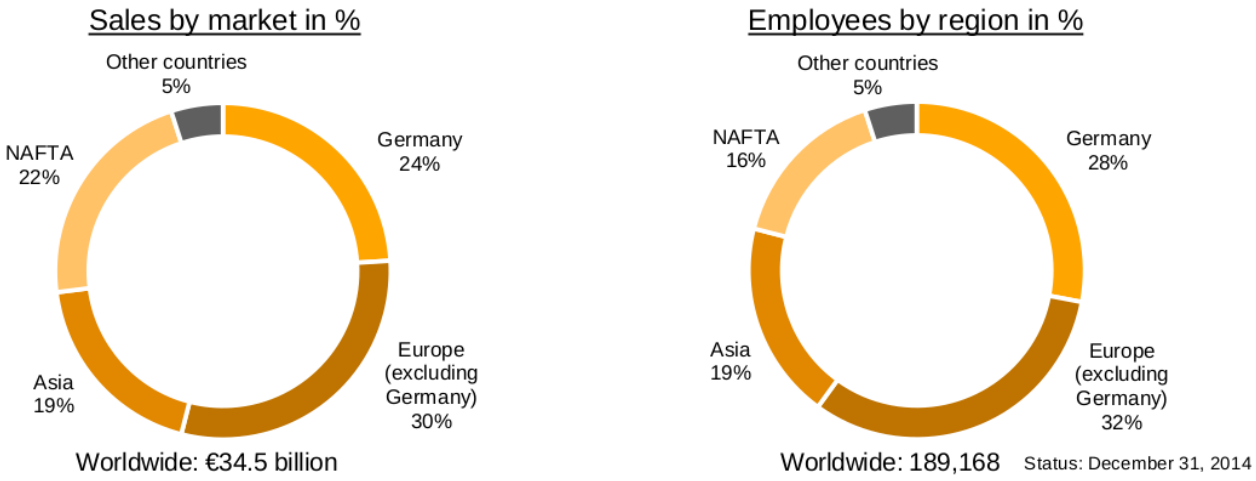


Figure 1.1 – Chiffre d'affaire et nombre d'employes (Annee 2014) [1]

1. NAFTA : North American Free Trade Agreement

En 2014, l'entreprise comptait plus de 189 000 employés dans le monde, comme le montre la figure 1.1 répartis dans 317 sites et 50 pays différents, dont la répartition est détaillée figure 1.2. Avec un chiffre d'affaire de 34.5 milliards d'euros au total, Continental est le numéro un du marché de production de pneus en Allemagne et est également un important équipementier automobile.

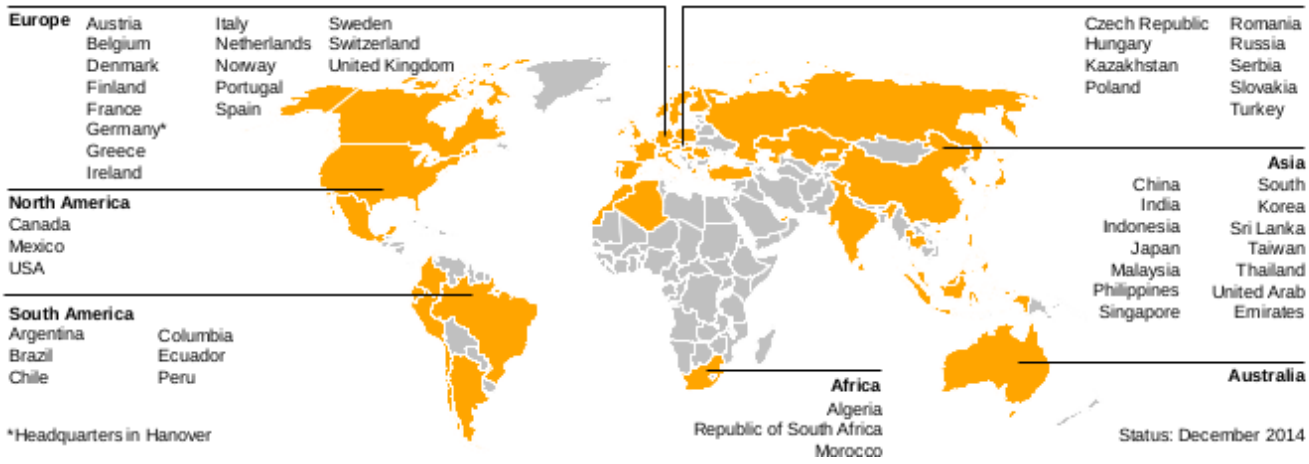


Figure 1.2 – Répartition du groupe Continental dans le monde

1.1.2 Histoire de l'entreprise

Continental est fondée en 1871 comme société anonyme sous le nom de « *Continental-Caoutchouc- und Gutta-Percha Compagnie* » par neuf banquiers et industriels de Hanovre (Allemagne).

Continental dépose l'emblème du cheval représenté sur la figure 1.3, comme marque de fabrique à l'Office impérial des brevets de Hanovre en octobre 1882. Ce logo est aujourd'hui encore protégé en tant que marque distinctive.

Figure 1.3 – Logo de Continental

Le fabricant de pneus allemand débute son expansion à l'international en tant que sous-traitant automobile international en 1979, expansion qu'il n'a cessé de poursuivre depuis.

Entre 1979 et 1985, Continental procède à plusieurs rachats qui permettent son essor en Europe, celui des activités pneumatiques européennes de l'américain *Uniroyal Inc.* et celui de l'autrichien *Semperit*.

En 1995 est créée la division « *Automotive Systems* » pour intensifier les activités « systèmes » de son industrie automobile.

La fin des années 1990 marque l'implantation de Continental en Amérique latine et en Europe de l'Est.

En 2001, pour renforcer sa position sur les marchés américain et asiatique, l'entreprise fait l'acquisition du spécialiste international de l'électronique *Temic*, qui dispose de sites de production en Amérique et en Asie. La même année, la compagnie reprend la majorité des parts de deux entreprises japonaises productrices de composants d'actionnement des freins et de freins à disques.

En 2004, le plus grand spécialiste mondial de la technologie du caoutchouc et des plastiques naît de la fusion entre *Phoenix AG* et *Conti'Tech*.

Enfin en juillet 2007, Continental réalise sa plus grosse opération en rachetant le fournisseur automobile *Siemens VDO Automotive*. Ce rachat a permis à l'entreprise de multiplier son chiffre d'affaire par 2.5, passant ainsi de 13 milliards d'euros à plus de 34.5 milliards d'euros (chiffre de 2014).

1.1.3 Activités des différentes branches

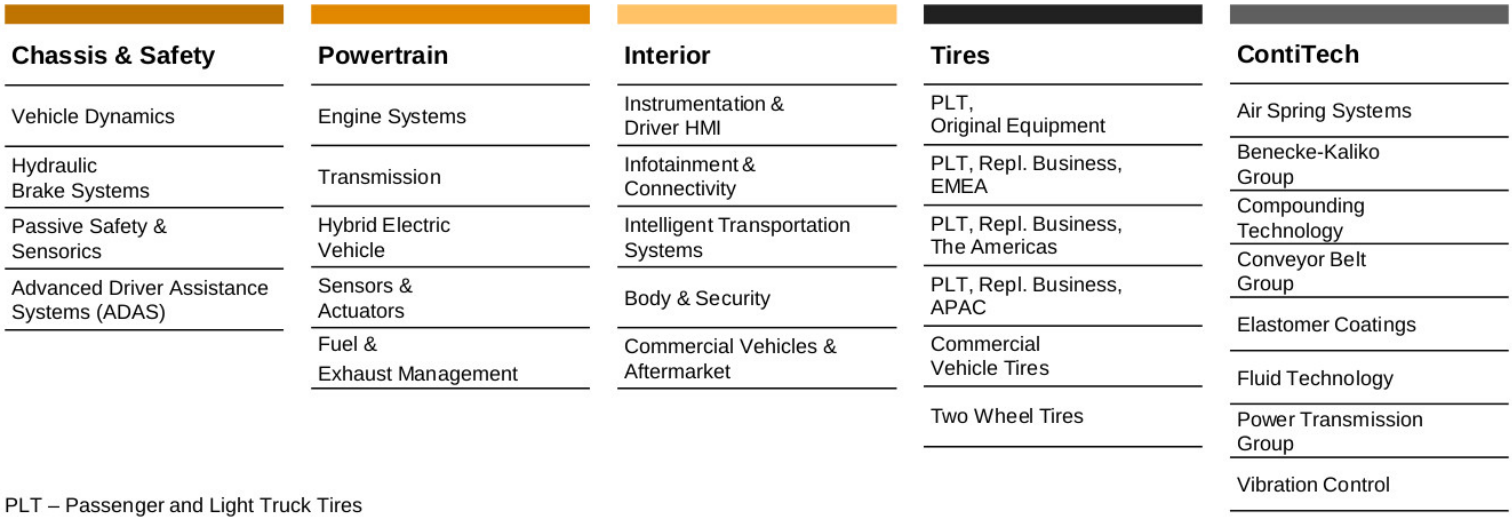


FIGURE 1.4 – Structure de Continental

Comme on peut le voir sur la figure 1.4, Continental est composée de cinq divisions. Ces dernières se chargent de développer et produire des équipements répondant aux besoins des clients. Pour cela elles sont composées de *Business Units* qui ont chacune une activité bien particulière dans leur domaine de compétence.

Durant mon stage, je travaillais au sein de *P-ES* :

Division *Powertrain* S'occupe essentiellement du contrôle moteur, au niveau logiciel et matériel avec l'ECU [2]

Business Unit *Engine Systems* Chargée de produire les équipements nécessaires au contrôle moteur tels que des calculateurs ou des injecteurs.

2. Engine Control Unit, Unité de calcul du contrôle moteur

1.2 Le contexte de l'équipe TAS

J'ai travaillé dans l'équipe en charge des tests au niveau système ou logiciel dirigée par Corinne Tarin. Cette équipe doit aider à la vérification et la validation des programmes de contrôle moteur en fournissant des services de tests.

1.2.1 Le besoin

Le calculateur du contrôle moteur d'une voiture est un dispositif très important et à haut risque, en effet, une défaillance peut provoquer la mort de plusieurs personnes [3]. Ainsi, le test est indispensable dans ce domaine, et doit être robuste.

Le test des logiciel de contrôle moteur se fait aujourd'hui :
- « Soit à la main » pour les tests d'intégration
- Soit à l'aide de scripts de test Python, écrit manuellement

Cependant, la taille des logiciels à tester est devenu particulièrement importante(Plusieurs milliers de variables, dans plus de 10 000 pages de spécifications...). Cela appel à une automatisation plus forte des tests afin d'augmenter fortement la vitesse et la quantité de tests afin d'éliminer le maximum de bogue.

C'est dans ce contexte que l'équipe TAS intervient, c'est ainsi que je participe au développement d'un outil permettant d'automatiser des tests d'intégrations pour les équipes projet travaillant pour Ford.

1.2.2 Les tests automatisés

Pour ma part, j'opérais dans la partie test automatique. Cette « sous-équipe » possède deux missions :
- Le développement et l'exécution de scripts de tests de non-régression [4]. Ces scripts de tests s'exécutent sur bancs HiL [5] avant la livraison des projets
- Le développement et la maintenance d'outils logiciels notamment en utilisant la TA3 présenté section 1.2.3. Ceux-ci permettent d'améliorer la couverture et la qualité des tests. Ils doivent permettre aux développeurs de vérifier facilement et correctement leur travail, particulièrement pour des tests de non-régression bien que l'outil sur lequel je travail soit à destination de tests d'intégration.

1.2.3 Les outils de tests

Afin d'effectuer son travail, l'équipe TAS possède différents outils de tests. D'une part au niveau matériel avec des bancs de tests, mais aussi logiciel avec une plateforme écrite en Python.

3. Le programme d'une voiture comporte ainsi des fonctions dites « *safety* » tel que le régulateur, l'accélération, le freinage, ...
4. Aussi appelés FaST : **F**unctions **a**nd **S**oftware **T**esting
5. **H**ardware **I**n the **L**oop. Vous trouverez plus d'explications sur ce dispositif section 1.2.3

Les bancs de tests

Afin de tester au mieux les programmes du contrôle moteur développés, ceux-ci sont d'abord testé via des simulateurs d'environnement véhicule. Ce simulateur permet de vérifier le programme avant d'effectuer des tests sur véhicule. Ces tests se font sur table dans un premier temps, pour deux raisons principales :

— D'un part, les tables sont plus facilement accessible qu'un véhicule d'essais pour les équipes logicielles

— D'autre part, les tables possèdent plus de moyens afin d'observer finement l'ECU

Comme vous pouvez le voir figure 1.5, un banc de tests est composé d'un ordinateur, du calculateur appelé ECU pour ***E**lectronic **C**ontrol **U**nit* et d'au moins deux équipements aidant aux tests.

Les deux équipements étant les suivants :

Le HiL Le ***H**ardware **i**n the **L**oop* est un simulateur d'environnement véhicule. Ainsi l'ECU est branché sur le HiL et se comporte de la même manière que s'il était embarqué dans une voiture. Le HiL quant à lui est chargé d'envoyer les bons stimulus sur les pins de l'ECU, tel que l'injection, la vitesse de rotation du moteur, le starter ...

Debugger Cet appareil est connecté au microcontrôleur de l'ECU via un port JTag. Il peut communiquer avec celui-ci afin d'effectuer différentes opérations. Tel que flasher le logiciel à tester, mettre des points d'arrêts sur le code, lire des variables, les modifier, changer des calibrations, ...

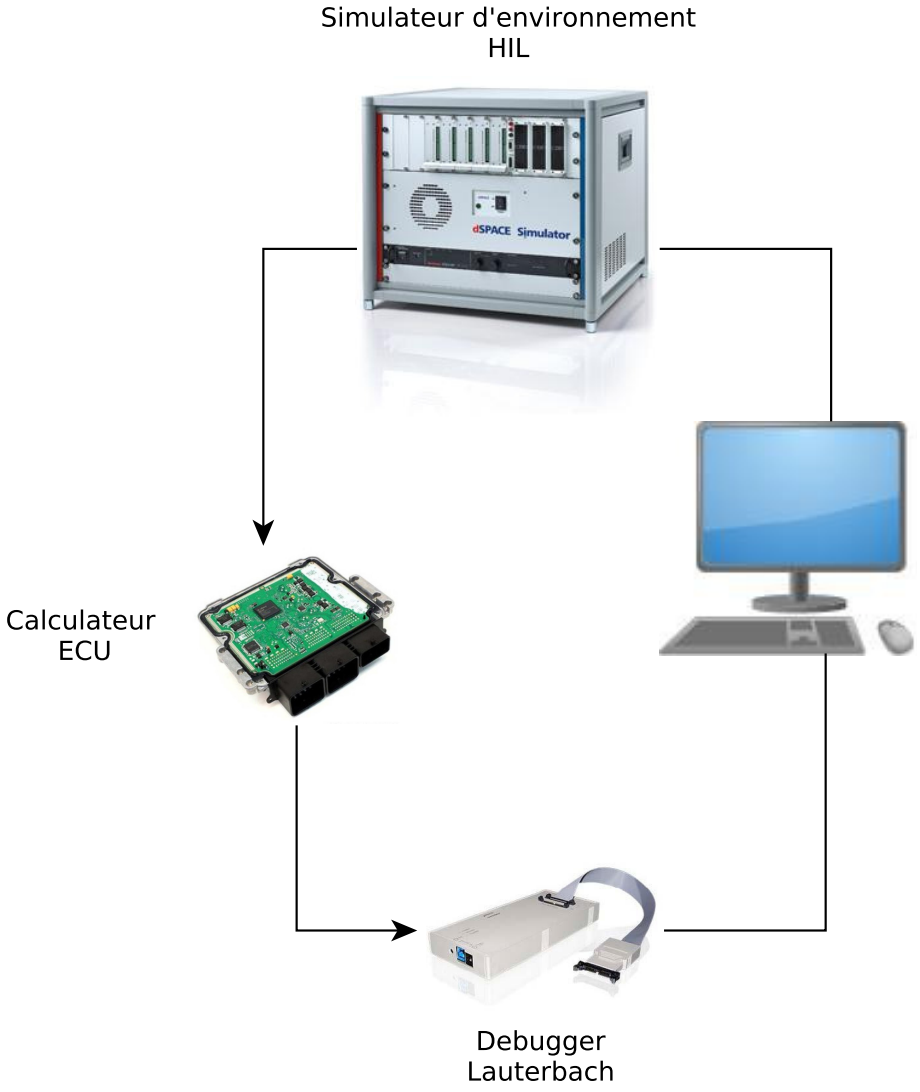


FIGURE 1.5 – Fonctionnement d'une table de tests : HIL DSpace, Debugger et ECU

Remarque Un certain nombre d'équipes projet chez Continental utilise un troisième équipement qui n'est pas représenté ici, parce que notre plateforme ne s'en sert pas. Cet outil, nommé INCA, va interagir sur l'ECU via un bus CAN [a].

a. Controller Area Network

Les différents équipements, ou *device*, que nous pouvons voir sur cette figure sont ceux avec lesquelles notre nouvelle plateforme va communiquer afin d'effectuer des tests automatiques.

La plateforme TA3

Actuellement, les équipes de tests disposent d'une plateforme appelée TA3. Celle-ci est une bibliothèque de classes écrites en Python. Jusqu'à présent, pour chaque objectif de test, il fallait écrire un script python utilisant la TA3. Ces scripts pilotent le banc HIL et le debugger afin d'envoyer des stimulis à l'unité de contrôle moteur et de vérifier que les réactions de celui-ci sont conforme aux spécifications de test.

Cependant, cette plateforme pose un certain nombre de problèmes qui rend son utilisation difficile. D'une part, elle renvoie un trop grand pourcentage de faux-positifs [6], faisant perdre du temps au testeur. D'autres part, elle ne prend pas en compte certains besoins apparu récemment comme par exemple un système permettant de flasher automatiquement les ECU [7], ou la possibilité de vérifier la fréquence de mise-à-jour de la production de variables [8].

Afin d'améliorer cette situation, l'équipe *Tests & Automation Service* développe une nouvelle plateforme.

6. Lorsque l'on effectue des tests, le but du testeur est de trouver des bugs. Ainsi, un positif est l'apparition d'un bug, et donc un faux positif signifie que la plateforme montre des bugs, qui sont inexistants
7. Cela permettrai de scripter un test qu'on lancerai plus tard et de gagner du temps
8. Ceci afin de contrôler le temps réel du calculateur

2 Le problème

Depuis longtemps, l'entreprise avait un problème afin d'effectuer des tests d'intégrations, notamment pour les projets à destination de Ford. Les tests demandaient du temps et de l'argent à l'équipe en charge de ces tests. Ainsi, deux ans avant mon stage de M1, une solution à été trouvée : le développement d'une nouvelle plateforme, *GreenT*.

Sommaire

2.1 Composant logiciel tiers

Étant donné la taille grandissante des programmes informatique, il est de plus en plus rare qu'une seule et unique entité effectue le développement d'un logiciel.

C'est ainsi que chez Continental, un certain nombre de composant des calculateurs ne sont pas développés en interne. Ce concept est appelé composant logiciel tiers, ou *Third-Party Software*.

La principale difficulté de ce mode de fonctionnement est l'intégration, une spécification exhaustive est indispensable afin de pouvoir connecter les différentes interfaces du composant avec le reste du projet.

C'est avec ces contraintes que travaillent plusieurs équipes chez Continental et notamment l'équipe en charge du développement des différents logiciels de contrôle moteur à destination de Ford.

2.2 Les tests du « plugin » Ford

Dans le cadre de projets pour Ford, Continental ne développe pas l'intégralité du logiciel, une partie étant fournie par le client sous forme de « plugin ». Ce plugin est un fichier binaire qui est chargé directement dans la mémoire flash de l'ECU sans que Continental n'ait accès au code source. Seule une description des interfaces est fournie. Le *plugin* est supposé correct, et le tester n'est pas de notre ressort. Cependant, celui-ci va être interfacé avec les logiciels Continental : il est indispensable de vérifier que les deux parties fonctionnent ensemble lors de l'intégration.

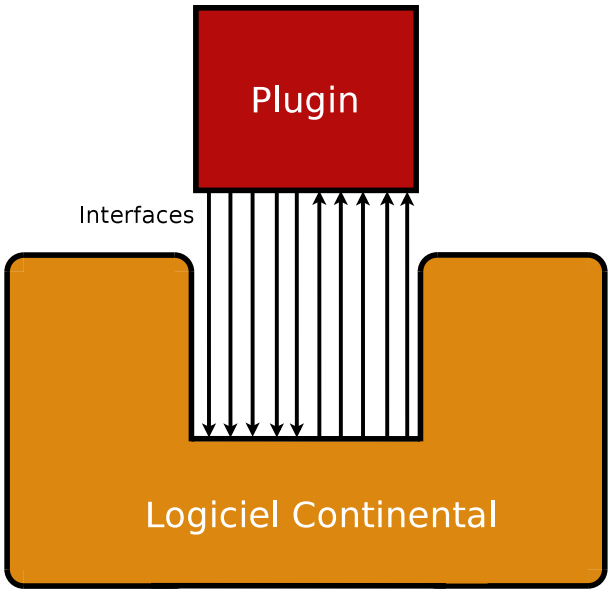


Figure 2.1 – Interfaces du plugin avec le logiciel de Continental

Le fichier de spécification est un fichier Excel, fourni par le client. Ce fichier, appelé `Walkthrough`[1], contient la liste de toutes les variables du plugin avec toutes leur spécifications. Il contient environ 900 variables différentes.

Il est impensable de tester le fonctionnement d'autant de paramètres manuellement. Ainsi l'équipe en charge de tester cette intégration effectue des tests de différence d'une version à l'autre : seules les variables ayant pu être impactées par une *release* seront testées, il est supposé que le fonctionnement des autres variables reste inchangé. Ce type de test est appelé *Delta Test*.

Deux problèmes se posent à cette méthode :

La fiabilité des tests manuels Le test des seules différences ne permet pas nécessairement de détecter tous les problèmes (notamment en cas d'effets de bords...). De plus, une tâche répétitive peut entrainer des erreurs humaines.

Le temps de tests Même en ne testant qu'une partie des variables, cela prend un temps considérable, il faut compter environ une semaine.
Or, les tests s'effectuent sur les bancs de tests comme expliqué section 1.2.3, ces équipements permettent de simuler un environnement voiture autour du contrôleur moteur comme l'utilisation de la clé de démarrage, la tension de la batterie, la vitesse de rotation du moteur, ... Ces bancs sont peu nombreux dans l'entreprise en raison de leur cout, leur disponibilité est compliquée. Il serait intéressant de pouvoir lancer des tests automatisés durant la nuit par exemple afin de les décharger de ce travail.

1. Ce fichier est expliqué plus en détail section 4.1

2.3 La solution : GreenT

Pour répondre aux besoins de l'équipe Ford, une solution a été pensée en étudiant leurs besoins : le développement de *GreenT*

> **Remarque** Le nom de *GreenT* provient de la contraction de *Green* et *Test*.
>
> En effet, un test est signalé correct si celui-ci est vert, or le but de notre plateforme est d'automatiser des tests et qu'à la fin de l'exécution, tout ceux-ci soient verts.

2.3.1 Génération de tests automatiques

Les tests d'intégration du plugin Ford

À court terme, cette solution devrait permettre de tester le plugin pour les projets Ford facilement et de façon efficace. Pour cela, les testeurs Continental vont ajouter des colonnes dans le document *Walkthrough*, afin de spécifier la manière de tester les variables. La plateforme, sera capable d'analyser le document *Walkthrough*, et de générer les tests automatiques. Le testeur pourra ensuite planifier son exécution, celle-ci devra prendre une dizaine d'heures pour 800 tests. Une fois l'exécution terminée, le testeur aura tous les résultats de ces tests, il pourra ainsi regarder les rapports détaillés afin de corriger les éventuels problèmes.

Ces tests s'effectueront sur des variables enregistrées lors de stimulation de l'ECU, afin de vérifier que celui-ci réagit de façon approprié.

Cette plateforme permettra donc de tester facilement la dizaine de projets Ford, et une fois le test d'une variable spécifiée, il n'est plus nécessaire de le réécrire. À chaque nouvelle version du logiciel, ou *release*, il suffira de relancer les tests : l'équipe n'aura à faire le travail qu'une fois, ensuite la réutilisation sera possible, les projets seront testés plus rapidement, plus efficacement, et plus souvent.

Les autres projets

À moyen terme, cette plateforme pourrait être utilisée pour les projets d'autres clients tel que Renault, afin d'effectuer là aussi des tests d'intégration. Il était nécessaire de concevoir une plateforme qui puisse évoluer facilement, et avoir un fichier de spécifications en entrée qui soit légèrement différent d'un client à l'autre.

En effet, les autres clients peuvent aussi fournir une partie du logiciel, avec un document de spécification des variables, celui-ci ne serait pas totalement identique, mais il s'en approchera.

Il est également envisageable que la plateforme soit utilisée pour des tests d'intégration en interne, indépendamment des spécifications fournies par un client externe à Continental.

2.3.2 L'utilisation de GreenT comme une bibliothèque

Une autre approche de notre plateforme, serait de s'en servir pour écrire facilement des tests en Java, de façon plus efficace et plus robuste qu'avec la TA3 : notre plateforme doit donc également fonctionner comme une bibliothèque sans utilisation de générateur ou de parser, pour que le testeur puisse effectuer un test rapide.

Celui-ci apprendra à se servir de la plateforme, écrira en règle générale des tests assez courts et moins complexes que ceux que nous générerons, ceux-ci doivent être faciles à écrire.

> **Remarque** *GreenT* fonctionne avec un modèle client-serveur comme cela sera détaillé chapitre 4.
>
> Afin de commencer à effectuer une livraison dans l'optique de l'utilisation de *GreenT* en tant que bibliothèque, nous avons déjà fourni les premières versions des serveurs à plusieurs personnes. Celles-ci utilisent nos serveurs en effectuant des scripts simples depuis un an sans problème.

3 Organisation du développement

Étant donné la complexité du projet et son importance, une organisation réfléchie est indispensable. Autant d'un point de vue humain, avec une gestion de projet et une gestion de l'équipe, que technique en utilisant certaines technologies nous aidant dans la tache. Nous allons voir l'organisation qui a été mise en place afin d'être le plus efficace possible.

Sommaire

3.1 L'équipe de développement

Au cours de mon stage, trois développeurs travaillaient sur le projet *GreenT* : Alain Fernandez, chef d'équipe et membre de l'équipe *Tests & Automation Service*, Benjamin Guerin, apprenti, et moi-même, stagiaire au sein de la même équipe.

En tant que chef d'équipe, Alain Fernandez organisait les réunions et supervisait notre travail tout en corrigeant des bogues, et validait manuellement les rapports de tests [1]. Benjamin Guerin se chargeait d'effectuer une étude de faisabilité sur l'amélioration visuelle des rapports de tests. Quant à moi je m'occupais de développer deux nouvelles fonctionnalités. [2].

À mon arrivée, la plateforme était quasiment opérationnelle, grâce à notre travail lors de mon précédent stage, et de l'avancement qui avait été fait durant cette année de césure. Je me suis donc d'abord renseigné sur les différentes améliorations et avancées du développement afin d'être rapidement opérationnel.

Ensemble, nous avons convenu de documenter au maximum notre travail, afin de conserver une plateforme toujours à jour au niveau de sa documentation. Ainsi pour chaque modification, chacun de nous devait remplir un document de *minutes d'étude*.

3.2 La documentation : les minutes étude

Je suis arrivé en Mai sur un projet ayant été commencé 18 mois auparavant, ainsi beaucoup de choses existaient déjà : il était nécessaire de garder l'existant. Afin de documenter notre travail, il

1. La validation manuelle des rapports est indispensable afin de vérifier que notre plateforme fournit des rapports fiables
2. Cf chapitre 5

a été décidé que pour chaque développement, que ça soit du bogue, de la fonctionnalité ou de la réorganisation du code, il était nécessaire de remplir un document Word.

Ce document comportait quatre grandes parties :

Analyse du besoin Pourquoi ce développement est nécessaire, les cas d'utilisation pris en charge, ou non, les éventuelles discussions avec l'équipe cliente.

Analyse de l'existant Retro-engineering permettant de comprendre le fonctionnement actuel du module que nous allons modifier, cette conception doit être statique, et dynamique, appuyé sur des schémas UML 2.

La solution Conception de notre solution, ses limites, ses avantages. De la même manière que la partie précédente, la conception est statique et dynamique, avec des schémas UML 2.

Les tests De quelle manière nous allons tester notre solution, aussi bien en tests unitaire qu'en tests d'intégration.

Ces différents documents pourront nous resservir plus tard pendant la maintenance, si un modèle doit être amélioré, ou comporte des problèmes, la relecture de la minute étude correspondante nous fera gagner beaucoup de temps.

3.3 Outils de développement

Afin de travailler de façon efficace, nous avons utilisés des outils aidant au développement. Ces outils ont été définis au début du projet, et n'ont pas évolués depuis.

3.3.1 Java

À mon arrivée, la partie client de notre plateforme était développée en Java dans sa version 6.0, Java nous permettant d'avoir un langage fortement typé, très puissant au niveau du paradigme Objet, connu de l'équipe, assez simple de déploiement et multiplateforme.

Une de mes collaboration a été le passage à Java 8 nous permettant d'utiliser toute la puissance de Java, et d'avoir une plateforme qui soit à jour au niveau technologique.

3.3.2 Git

Nous avons utilisé *Git* afin de faciliter le travail collaboratif d'une part, et de versionner le code du logiciel d'autres part. Git permet de fusionner les modifications de plusieurs développeurs, tant que nous ne modifions pas le même fichier en même temps. Ainsi, la fusion de nos modifications était faite automatiquement.

De plus, à chaque nouvelle modification, un « commit », permet de créer un point de restauration : il est alors possible de récupérer n'importe quelle version du logiciel depuis son commencement. Nous y insérons un message clair expliquant ce qui a été fait, cela permet aux autres développeurs de l'équipe de se tenir au courant de l'avancement.

3.3.3 Thrift et client-serveur

Notre plateforme fonctionne avec une architecture client-serveur, un client et deux serveurs. Le client écrit en Java, un serveur utilise Python et le second est lui aussi en Java. Afin de faire communiquer les deux parties de notre application, nous avons utilisé *Apache Thrift*. Il s'agit d'une bibliothèque ayant pour but les communications réseau inter-langage, dans le même principe que le protocole RMI [3].

Ainsi, nous avons rédigé un fichier spécifiant les interfaces de notre serveur, c'est-à-dire les méthodes que nous souhaitions appeler en réseau. Une fois ce « contrat » rédigé, il faut demander à Thrift de générer le code du serveur [4], et du client. Côté client, le service s'utilise directement, côté serveur, il faut implémenter une interface afin que notre service effectue les bonnes instructions. C'est donc le code généré qui va se charger de l'abstraction réseau.

3.3.4 Eclipse

Nous développions tous sous le même environnement de développement Eclipse, avec le plugin *Git* et le plugin *PyDev*. Le plugin Git permet d'avoir des outils aidant à la résolution d'éventuels conflits et le plugin PyDev permet de développer avec l'interpréteur et la coloration syntaxique Python.

3.3.5 Antlr

Pour les besoins de notre plateforme, nous avons créé notre propre langage de test. Ce langage est assez riche, et la création d'un parser adéquate aurait pu être particulièrement long. Afin de nous faire gagner le maximum de temps, nous avons utilisé Antlr4. *Another Tool For Language Recognition* est un programme permettant de générer automatiquement un parser pour un langage donné. Ainsi, nous avions rédigés notre grammaire, Antlr quant à lui s'est chargé de nous générer un arbre de parcours syntaxique. À notre charge d'effectuer les bonnes actions durant le parcours de notre langage en spécialisant les classes générées par Antlr.

3. Remote Method Invocation
4. Il est possible de demander la génération en C, C++, Python, Java, C#, PHP, Ruby, ...

3.3.6 UML et Entreprise Architect

Nous avons travaillé avec la norme UML [5] 2 afin de concevoir la plateforme, en utilisant particulièrement des diagrammes de classes, mais aussi des diagrammes de cas d'utilisation ou d'activité.

Pour dessiner ces diagrammes, et les noter dans la documentation, nous les pensions d'abord sur tableau blanc, mais ensuite nous avions besoin d'un outil puissant afin de les dessiner sur informatique. Pour cela nous avons utilisé *Enterprise Architect*, un logiciel propriétaire permettant de créer tous les diagrammes de la norme UML 2.

3.3.7 SQLite

SQLite est un moteur de base de données relationnelle. Sa particularité est de ne pas reproduire le schéma habituel client-serveur mais d'être directement intégré aux programmes, la base de données étant stockée dans un simple fichier.

Nous nous en sommes servis afin de pouvoir stocker les différentes informations d'un test, ceci afin de pouvoir redémarrer une exécution grâce à cet état intermédiaire conservé en base de données.

3.3.8 LaTeX

Afin de rédiger ce rapport, et le diaporama de soutenance, j'ai utilisé LaTeX, un langage et un système de composition de documents fonctionnant à l'aide de macro-commandes. Son principal avantage est de privilégier le contenu à la mise en forme, celle-ci étant réalisée automatiquement par le système une fois un style défini.

LaTeX

5. Unified Modelling Language

4 *GreenT* : fonctionnement général

Comme nous l'avons montré dans le chapitre 2, l'entreprise a besoin d'un nouvel outil aidant aux tests d'intégration : *GreenT*.

Nous allons donc voir le développement et la conception de cette plateforme de tests.

Sommaire

Au début de mon stage, le projet ayant un an, les fonctionnalités développées ci-dessous étaient déjà faites. Je suis cependant intervenu sur la plupart d'entre elles soit pour des corrections de bogues d'une part, soit à des fin d'améliorations d'autre parts.

Avant de présenter mon travail, que vous trouverez chapitre 5, il est nécessaire de présenter le fonctionnement général de cette plateforme afin d'en avoir une vue d'ensemble.

4.1 Le fichier Walkthrough

Le fichier *Walkthrough* est un fichier qui sera fourni par la personne en charge des tests, c'est un fichier au format Excel qui contient les informations de chacune des variables à tester. Il contient ainsi un très grand nombre de colonnes, bien que seule une partie de celles-ci nous intéressent. Certaines colonnes ont été remplies par le fournisseur du plugin, d'autres colonnes sont ajoutées dans le seul but de la génération de tests automatiques par *GreenT*. Voici les informations les plus intéressantes :

Nom de la variable Le nom de la variable testée : il existe un nom court et un nom long.

Informations aidant à la conversion des données Certains équipements [1] tel que le *debugger* ne fonctionne qu'avec des valeurs Hexadécimales. À la charge de *GreenT* de convertir ces données vers des valeurs physiques exploitables par le testeur. Ces colonnes contiennent les informations nécessaires au calcul de conversion [2].

Nécessité d'un test automatique Un `GreenTTest` ne sera généré que si la colonne vaut *Yes*.

Statut du test La plateforme éditera automatiquement cette colonne afin de reporter le statut

1. Vous trouverez plus d'informations sur le fonctionnement d'une table de tests section 1.2.3
2. Informations tel que le domaine de définition physique et le domaine de définition hexadécimal, avec ces deux informations il est possible d'effectuer les conversions physique vers hexadécimal

du test [3].

Précondition (cf section 4.3.2) Contient un scénario d'initialisation du *workbench* : tension de départ, démarrage de l'ECU, ...

Scénario de stimulation (cf section 4.3.2) Contient un ou plusieurs scenarii de stimulations destinés à faire générer au HiL un certain nombre de stimuli.

`ExpectedBehavior` (comportement attendu, cf section 4.3.3) Contient une expression évaluant les variables ayant été enregistrées durant la stimulation : *GreenT* devra vérifier que cette expression est correcte à toute instant de la stimulation.

Variable à enregistrer (cf section 4.3.3) Contient les variables devant être enregistrées durant un scénario, en plus des variables présentes dans l'expected behavior. Celles-ci peuvent servir en tant que données contextuelles permettant de mieux cerner le résultat d'un test.

Informations du test (cf section 4.3.5) Plusieurs colonnes telles que la sévérité, le responsable du test, des commentaires, ...

	RB_Variable	Test_InterfaceTy	Test_Conditi	Test_Stimulus	Test_ExpectedBehavior	Test_RecordedVariab	Test_LocalAli	Test_Sevel
2	ACCIntVlv_iSens	IO	SUB_ECU_GO	SCENARIO IO_VACC_control1 SUB_SCE_BENCH_ACC_0_100_0_POURC_Acti veBenchMode_0 END SCENARIO	EVAL(ACCIntVlv_iSens=(v_pwm_cfb_acc_slv / NC_C			Low
3	ACCIntVlv_r	IO	SUB_ECU_GO	SCENARIO wait2s SUB_WAIT_2S END SCENARIO	EVAL(pwm_ducy_acc_slv_cus=ACCIntVlv_r , TOLRES			Low
74	Air_uRawTEGRClrDs	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(Air_uRawTEGRClrDs=0)			Low
100	BrkBstP_uRaw	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(BrkBstP_uRaw=0)			Low
117	Clth_b25PrcSens	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(Clth_b25PrcSens=0)			Low
133	Clth_uAnaRaw	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(Clth_uAnaRaw=0)			Low
330	EGRVlv_uRaw	IO	SUB_ECU_GO	SCENARIO IO_EGRV_control SUB_SCE_EGRV_POS_0_5_0_V END SCENARIO	EVAL(EGRVlv_uRaw=v_egrv_mes[0] *1000, TOLRES			Low
357	EnvP_pSens	IO	SUB_ECU_GO	SCENARIO IO_AMP_control SUB_SCE_AMP_0_5_0_V END SCENARIO	EVAL(EnvP_pSens=amp_mes , TOLRES(1))			Low
362	EnvT_tSens	CALCULATION	SUB_ECU_GO	SCENARIO IO_TAM_control SUB_SCE_TAMB_0_5_0_V END SCENARIO	EVAL(EnvT_tSens=temp_air_mes[2] , TOLRES(1))			Low
363	EnvT_uRaw	IO	SUB_ECU_GO	SCENARIO IO_TAM_control SUB_SCE_TAMB_0_5_0_V END SCENARIO	EVAL(EnvT_uRaw=vp_temp_air[2]*1000 , TOLRES(1)			Low
389	Epm_stEposCPF	CALCULATION	SUB_ECU_GO	SCENARIO CALC_N_500_control SUB_SCE_N_0_500_0_RPM END SCENARIO	IF (lv_first_vld_tooth=0) THEN EVAL(Epm_stEposCPI			Low
422	Exh_uRawTOxiCatDs	IO	SUB_ECU_GO	SCENARIO IO_TEMP_UP_CAT_control SUB_SCE_TEG_UP_CAT_0_0_5_0_V END SCENARIO	EVAL(Exh_uRawTOxiCatDs=vp_teg_sns_avl[2] , TOLI			Low
426	Exh_uRawTPFltDs	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(Exh_uRawTPFltDs=0)			Low
439	FlSys_stDeflate	CALCULATION	SUB_ECU_GO	SCENARIO wait2s SUB_WAIT_2S END SCENARIO	EVAL(FlSys_stDeflate=state_fuel_pipe_ex_air)			Low
441	Fl_uRawFlFWLvl	IO	SUB_ECU_GO	SCENARIO IO_WFS_control SUB_SCE_WFS_0_5_0_V END SCENARIO	EVAL(Fl_uRawFlFWLvl = vp_fuel_warn , TOLRES(1))			Low
464	Fan_rPs	CALCULATION	SUB_ECU_GO	SCENARIO SCE_CFA_MAN_1 SUB_SCE_CFA_MAN_1_0 END SCENARIO	IF (lv_cfa_2=1) THEN EVAL(Fan_rPs=100)ELSE Fl			Low

Figure 4.1 – Aperçu d'un fichier Walkthrough

3. Un test pouvant être *Green* ou *Red* mais peut aussi comporter une erreur, tel qu'un problème d'exécution ou de génération, ...

4.2 Fonctionnement Global

Le développement de *GreenT* inclut un certain nombre de fonctionnalités attendues par le client et indispensable à son fonctionnement. D'autres fonctionnalités pourront apparaître plus tard en fonction des besoins.

Les principaux modules sont les suivants, avec leurs interactions schématisées figure 4.2 : dans des objets ovales sont représentés des fichiers, les carrés représentent des modules de la plateforme et les flèches en pointillés un transfert réseau, les couleurs représentent les différents modules de la plateforme.

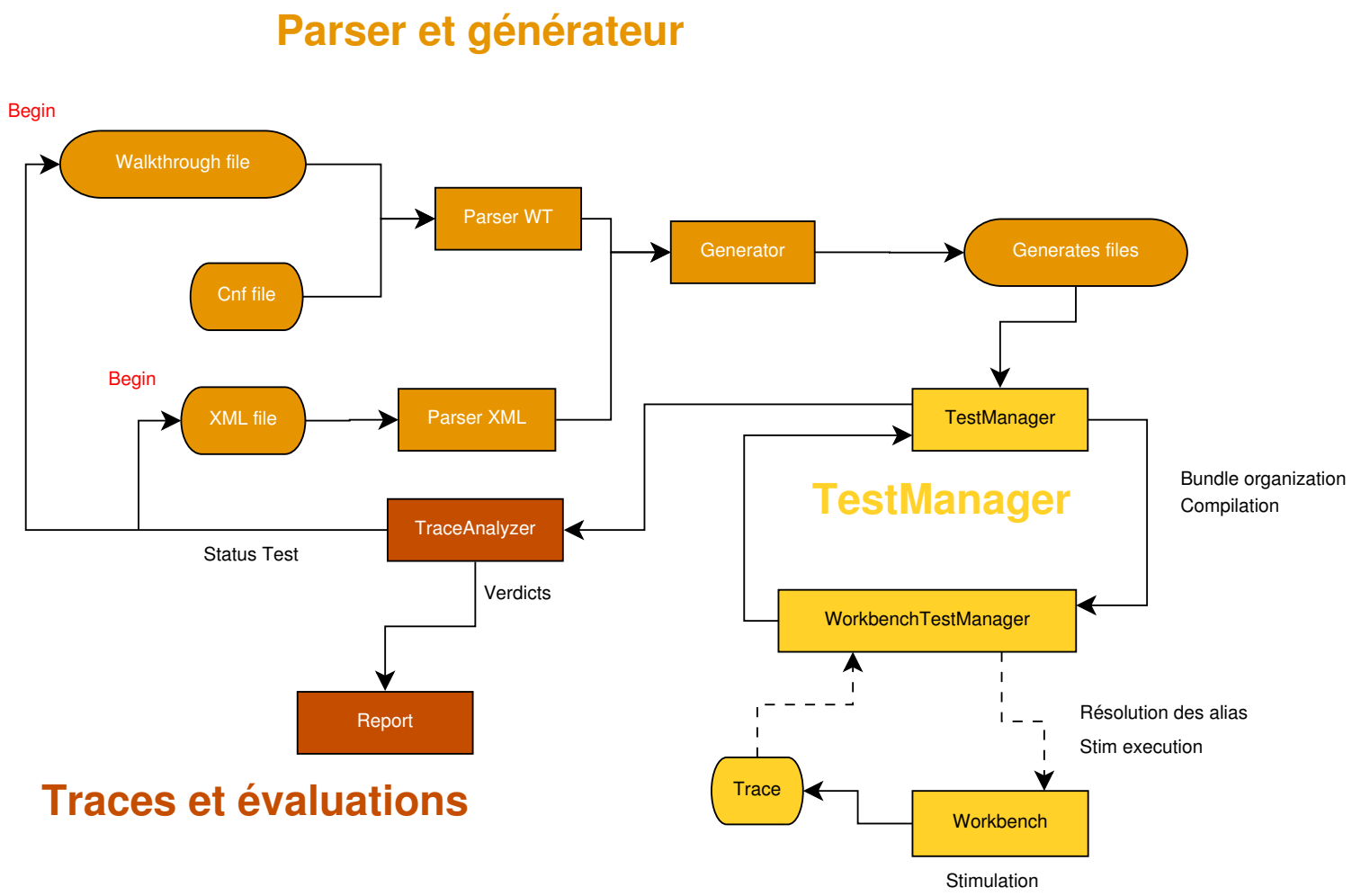


FIGURE 4.2 – Fonctionnement général de la plateforme *GreenT*

Dans la figure 4.2, vous pouvez voir des flèches en pointillés représentant des échanges réseau. En effet l'exécution des stimulations et l'enregistrement des traces se fait par un contrôle distant des bancs de tests [4].

Les principales fonctionnalités demandées par le client sont disponibles sur le client, la majorité peuvent s'effectuer en local :

- Le parsing et la génération (section 4.3.1)
- L'organisation en Bundle afin d'optimiser le temps d'exécution des tests (section 4.3.4)

4. La schématisation du fonctionnement d'un banc est disponible section 1.2.3 figure 1.5

— La production de rapports détaillés (section 4.3.3)

Alors que d'autres fonctionnalités vont nécessiter la présence de serveurs et de connexion réseau permettant d'effectuer ces actions :
— Les stimulations (section 4.3.2)
— Les enregistrement des traces (section 4.3.2)

4.3 Les fonctionnalités du Client

Afin de répondre au mieux aux attentes du client, il a été choisi d'utiliser le langage Java pour développer notre client, ceci en raison de divers avantages comme le côté multi-plateforme, son typage fort et sa forte communauté qui permet ainsi une maintenance facilitée.

4.3.1 Parsing et Génération

Le but premier de la plateforme est d'effectuer des tests automatiques, il est ainsi indispensable d'avoir un système d'automatisation, au travers d'une génération.

Pour cela, nous avons un parser : il analyse un certain type de fichier [5] et en retire pour chaque test, le scénario de pré condition, les différents scénarios de stimulations, leur *Expected Behavior*, les données qui devront être enregistrées ainsi que différentes informations sur le test [6].

Une fois toutes ces données acquises, il les transmet à un générateur qui est en charge d'écrire les fichiers Java de chaque test, tous sont organisés dans un dossier temporaire avec un dossier par test. Le `TestManager` peut ensuite traiter ces données.

Exemple Le testeur souhaite vérifier le bon fonctionnement des ventilateurs de refroidissement du moteur. Ci-dessous les différentes cellules qui pourrait être renseigné par le testeur pour ce cas de tests. Nous verrons ce qu'effectuent précisément ces actions dans la suite de la section.

Précondition	Scénario de Stimulation	Expected Behavior

`// Tension batterie`
`HIL_VB = 13;`
`// Clé moteur`
`HIL_KEY = 1;`
`// Vérifications`
`CHECK(HIL_KEY = 1`
`  && HIL_VB = 13);`
	`// Rampe de vitesse véhicule`
	`// de 0 à 50km/h par pas de 1`
	`// toutes les secondes`
	`RAMP(HIL_VS, 0, 50, 1, 1);`
		`if temperature > max then`
		`  // Ventilateur allumé`
		`  EVAL(ventilateur_on = 1);`
		`else`
		`  // Ventilateur éteint`
		`  EVAL(ventilateur_on = 0);`
		`end if`

4.3.2 Stimulation

Afin de tester une variable du plugin, les développeurs vont utiliser des alias présents sur un device : actuellement, un HIL ou un debugger, prochainement nous pourrions en utiliser d'autres. Ces alias permettent de simplifier le travail du spécifieur, il n'aura pas à retenir une adresse d'un élément sur le HiL, un simple alias permet de lui faire cette abstraction et ce raccourci.

5. Nous ne commencerons qu'avec le Walkthrough pour débuter, mais dans le futur nous pourrions avoir des fichiers XML, des bases de données, ...
6. Responsable du test, sévérité, commentaires, nom de la variable, ...

Le spécifieur va rédiger des scénarios de stimulation, ceci afin de mettre le contrôleur dans certaines conditions. Son but sera ensuite de vérifier que ces variables restent cohérentes vis-à-vis du scénario effectué.

Un scénario particulier doit être spécifié : une pré condition qui a pour but d'initialiser les équipements et certains alias afin d'avoir un état de stimulation qui soit cohérent et identique à chaque lancement du scénario. Ce scénario sera effectué avant le lancement de chacun des scénarios de stimulation.

Durant l'exécution d'une stimulation, les variables nécessaires sont enregistrées afin de pouvoir produire des rapports et des verdicts ensuite.

Exemple Comme nous l'avons vu section 4.3.1, le testeur nous a donné un scénario de stimulation ainsi qu'un scénario de précondition. Le code généré va ainsi communiquer avec le serveur HIL afin qu'il envoie les bons stimulis à l'ECU.

Ainsi, comme mis en commentaires, le scénario de précondition va mettre la batterie à 13 Volts, et mettre la clé, nous allons ensuite vérifier que cette action a bien été fait. Si tel est le cas, le scénario de stimulation va être effectué. Ce scénario va effectuer une rampe de vitesse, notre véhicule va aller de zéro à cinquante kilomètre-heure avant de retourner à l'arrêt.

4.3.3 Les traces et leurs évaluations

Lorsqu'un scénario de stimulation s'exécute, un certain nombre de variables sont enregistrées : ces variables sont stockées sous la forme d'une trace au format CSV [7], qui pourra plus tard être représentée sous forme de courbe.

Une fois que la trace est complète, il est nécessaire de l'évaluer : le spécifieur a décrit le comportement attendu dans la colonne *Expected Behavior* détaillant dans quel cas le test est correct. Cette expression va être transformée en arbre logique afin de l'évaluer à tout instant de la trace.

Exemple Durant notre rampe véhicule, le *debugger* a enregistré différentes variables, en l'occurence nous avons enregistré les trois variables présentes dans notre *Expected Behavior* : la température du moteur – `temperature`, la température maximum sans ventilateur – `max`, ainsi que l'état de notre ventilateur – `ventilateur_on`.

Ces variables ont été enregistrées durant l'intégralité de la rampe véhicule qui à eu une durée de cinquante secondes. À la fin de la stimulation, le serveur retourne ainsi une trace de cinquante secondes contenant toutes les variables. *GreenT* peut ensuite évaluer notre *expected behavior* sur l'intégralité de l'enregistrement.

4.3.4 Le module TestManager

Comme le montre la figure 4.2, la classe `TestManager` est le chef d'orchestre de *GreenT*, il a donc un certain nombre de responsabilités.

Il va d'abord organiser les différents tests en un concept que nous avons appelé *Bundle*, ceci dans un but d'optimisation du temps d'exécution. En effet, si nous avons 1000 tests de 2 minutes, cela ferait plus de trente heures d'exécution. Pour palier à ce problèmes, nous avons deux stratégies :

7. Comma Separated Values

— Le regroupement de tests ensemble, si deux tests possèdent le même scénario de stimulation, alors nous n'executerons qu'une seule fois ce scénario, et évaluerons la trace pour chacun des tests. Ce regroupement est appelé «Bundle».
— Même avec notre stratégie des Bundles, l'exécution pourrait être encore trop longue. Ainsi, il est possible d'utiliser plusieurs tables en simultanées. Le temps d'exécution est ainsi divisé par le nombre de tables.

Afin d'être le plus souple possible, il existe plusieurs modes d'exécution de la classe `TestManager` :

Check only Essaye de parser les différents fichiers, et vérifie que ceux-ci ne comportent aucune erreur de grammaire, d'alias introuvable, d'écriture sur un alias en lecture seule etc... Cela permet à l'auteur des cas de test de se vérifier sans avoir besoin de table de test.

Parse and generate bundles Parse les fichiers et génère des jars exécutables répartis en bundle

Parse and execute Parse les fichiers, génère les jars pour les bundles et les exécute : c'est le mode « classique ».

Restart test execution Redémarre une exécution qui se serait mal terminée. Ceci à l'aide d'une base de données SQLite. Cette base de données contient toutes les informations des différents scénarios et sera capable de redémarrer à l'endroit où une coupure à eu lieu. Cela évite de devoir effectuer de nouveau trente heure d'exécutions si le problème à eu lieu sur la fin.

4.3.5 Production de rapport détaillé

La plateforme a en charge la production d'un rapport détaillé pour chaque test. Ce rapport contiendra un certain nombre d'informations, et permettra au testeur de comprendre pourquoi le test n'est pas passé. Voici les informations que contiendra ce rapport :

— Nom du test, de la variable à tester
— Nom du responsable du test
— Sévérité du test
— Pourcentage de branches de l'expectedBehavior renvoyant faux(Test « Rouge »), n'ayant pas pu être testé(Test « Gris ») et étant correct(Test Vert)
— Le testeur aura à sa disposition les expressions concernées par un résultat Rouge ou Gris.
— Les colonnes utiles du `Walkthrough`

Actuellement, les rapports se font au format Excel avec l'intégralité de notre enregistrement et pour chaque instant d'enregistrement(timestamp), un verdict. Un exemple de rapport est accessible en Annexe C page 51.

Dans un futur proche, ces rapports pourraient être générés dans un format Web avec une possibilité de naviguer entre plusieurs tests, et d'avoir un affichage des courbes de manière graphique.

4.3.6 Mise à jour du Walkthrough

Une fois l'analyse d'un test exécuté, un verdict global est mis dans le fichier *Walthrought* : ce verdict est consolidé en fonction des rapports détaillés. Ainsi, un test sera vert si l'expression a été validée sur l'ensemble de la trace, le test sera rouge dans le cas contraire.

Remarque En cas d'erreur à la génération [a] ou à l'exécution [b], la plateforme doit afficher un message d'erreur clair au niveau du test afin que l'utilisateur soit conscient du problème. Libre à lui de corriger le test si nécessaire, ou de le signaler si cela semble être un bogue. Ce message d'erreur est reportée automatiquement dans le *Walkthrough*.

a. Mauvaise syntaxe, variables inexistantes, ...
b. Problèmes réseaux, variable non trouvée, communication entre l'ECU et le debugger, ...

4.4 Le fonctionnalités des serveurs

Comme expliqué précédemment, *GreenT* va avoir en charge l'exécution de stimulation. Ces stimulations vont communiquer avec une table de tests. Actuellement une table est composée de deux équipements différents, comportant chacun leur serveur :

— Un HIL, Hardware In the Loop, simulateur d'environnement véhicule
— Un Debugger permettant de voir l'état du programme présent dans l'ECU

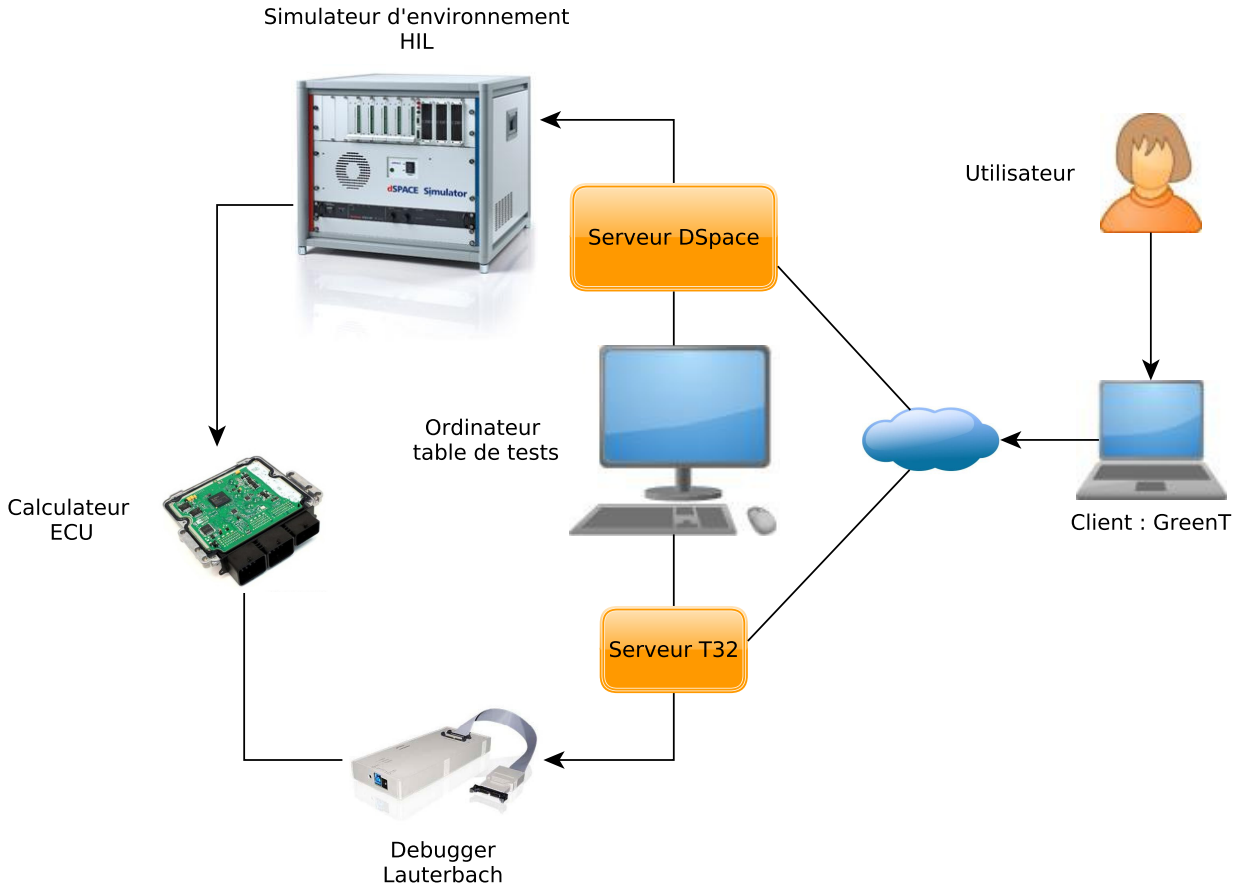


FIGURE 4.3 – Communications de la plateforme

Au début du développement de la plateforme, il a été décidé que les serveurs devront être le plus simple possible pour plusieurs raisons :

— Donner accès à un maximum de fonctionnalités en mode « offline », c'est-à-dire sans accès à une table de test
— Rabattre le maximum de fonctions métier près du client pour centraliser au maximum le fonctionnement et éviter la maintenance superflue
— Avoir la possibilité de réutiliser les serveurs pour d'autres projets

— Pouvoir ajouter facilement un nouveau *device*, qui ne nécessiterai que l'ajout d'un nouveau serveur relativement simple

4.4.1 Le serveur Debugger, contrôle de Trace32

Le serveur Debugger propose des services basiques permettant de répondre aux besoins :
— Flasher le logiciel dans la flash de l'ECU
— Démarrer l'ECU
— Arrêter l'ECU
— Lire une variable ou une calibration
— Modifier une variable
— Enregistrer des variables

Afin d'effectuer ces actions, le serveur s'appuie sur une API fournie par l'outil Trace32 permettant de contrôler le debugger. Ainsi tous nos services vont s'appuyer sur cette API. C'est pour cette raison que ce serveur est développé en Java afin de pouvoir utiliser facilement ces fonctions.

4.4.2 Le serveur HiL, contrôle du ControlDesk DSPace

Le HiL contient une base de données, représentant un modèle simulant l'environnement véhicule. Ce modèle à pour but de se rapprocher au maximum du fonctionnement réel de notre moteur.

Le serveur DSpace va devoir lui aussi répondre aux différentes stimulations, ainsi ces services sont relativement similaire :
— Modifier une valeur du modèle
— Lire une valeur du modèle
— Enregistrer des valeurs

Ce serveur a été développé en réutilisant une partie de ce qui avait été fait pour la TA3, présenté section 1.2.3 afin de ne pas « réinventer la roue ».

À l'instar du serveur Debugger, nous utilisons une API fournie par l'outil permettant de contrôler le HIL : ControlDesk. C'est ainsi que ce serveur est développé en Python afin de répondre à cette contraintes : l'API du ControlDesk est en Python. Ainsi, l'utilisation de Thrift, permettant de faire dialoguer du Java côté client et du Python côté serveur facilement, nous aura particulièrement aidé.

5 Ma collaboration au projet

Après avoir défini plus en détails les besoins de notre plateforme et son fonctionnement général, nous allons maintenant voir en détail de quelle manière j'ai contribué à ce projet. En parallèle de la maintenance de notre plateforme, j'ai développé deux nouvelles fonctionnalités. Ces deux fonctionnalités ayant un rapport direct avec la notion de « calibration », nous allons tout d'abord définir celles-ci avant de voir en détail mon développement et la maintenance que j'ai effectué.

Sommaire

5.1 Les calibrations

Une calibration est une constante stockée en flash, c'est-à-dire en mémoire non volatile. Ainsi le logiciel du contrôle moteur peut accéder à toutes ces calibrations en lecture uniquement.

Ces calibrations permettent de configurer un véhicule avant sa mise en production, permettant la variabilité du logiciel. Elles permettent plusieurs choses :

- Configurer un calculateur pour tous les types de véhicule qu'il doit supporter. Certains ECU couvrent depuis l'entrée de game jusqu'au milieu de game d'un ou plusieurs constructeurs[1]. On peut configurer la présence ou l'absence de turbo, la taille des roues, le nombre d'injecteurs... Autant de paramètre qui influent sur les lois de contrôle implantées dans l'ECU
- Ajuster le fonctionnement d'une loi de contrôle sur banc moteur ou sur véhicule pour tenir compte des normes de polution par exemple
- Traçabilité du calculateur en cas de retour fournisseur, ceci via des informations logistiques(Numéro de série, version du logiciel, ...)

Une fois le logiciel d'un calculateur mis en production, ces calibrations ne doivent pas évoluer, les modifier ne sert donc qu'à la mise au point et à la généricité du logiciel.

1. Renault, Niassan et Dacia par exemple

5.2 Le « patch calib »

Pour un certain nombre de scénario de stimulation, il est nécessaire de modifier des valeurs de calibrations. Cette action demande d'ajouter une grammaire spécifique : `PATCH_CAL(calibration, valeur)`. En interne, cela générera du code permettant de flasher la nouvelle valeur.

5.2.1 Expression du besoin

Les cas d'utilisation

Comme nous pouvons le voir figure 5.1, le patch de calibrations est répertorié en deux grandes parties : le parser avec la génération des instructions permettant de changer la valeur d'une calibration, et l'exécution du patch proprement dit.

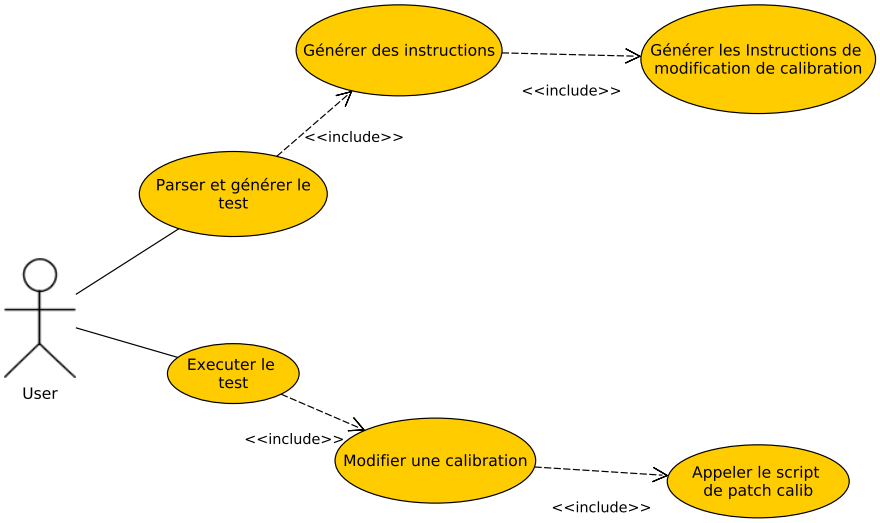


Figure 5.1 – Diagramme de cas d'utilisations du patch de calibrations

Limitations du patch calib

Afin de concevoir au mieux la solution répondant aux besoins du client, j'ai tout d'abord du me renseigner sur la manière dont je pouvais effectuer cette action sur l'ECU, c'est-à-dire l'implémentation du serveur. Après discussions avec les personnes compétentes, une restriction impactant mon développement a été mis au clair.

Afin de modifier une calibration, il est nécessaire de modifier la mémoire flash. Cette modification va nous obliger à éteindre l'ECU, la modification de la flash étant impossible ECU on. Il ne sera donc pas possible de modifier une calibration au milieu d'un scénario de stimulation, le scénario n'aurait aucun sens si nous éteignons et rallumons l'ECU pendant celui-ci.

Cette restriction n'a cependant pas d'impact sur les tests qui seront rédigés, un scénario de stimulation a pour but de se rapprocher du fonctionnement nominal d'une voiture, or il est inconcevable de modifier une calibration pendant que la voiture tourne. Du point de vue des tests, ce choix reste cohérent.

5.2.2 Fonctionnement à l'utilisation

Comme vu section 5.2.1, le « patch calib » ne peut être utilisé que si l'ECU est arrêté. Afin de forcer ce fonctionnement, et d'éviter une mauvaise utilisation, il a été choisi d'utiliser la grammaire du langage. En effet, une mauvaise utilisation de la fonctionnalité nous renverra une *syntax error*, forçant l'utilisateur à corriger cela.

```
SCENARIO NomDuScenario
  PATCH SECTION
    patch_cal(cal1, value1);
    patch_cal(cal2, value2);
  END SECTION
  // Stimulation proprement dite
  // Rampe, wait, ...
END SCENARIO
```

Extrait de code 5.1 – Scénario de stimulation contenant des patchs de calibration

L'utilisateur à la possibilité d'écrire plusieurs scénarii dans un même test, avec cette même syntaxe. Cette fonctionnalité permettrai à un testeur de vérifier différentes stratégies en fonction de calibrations.

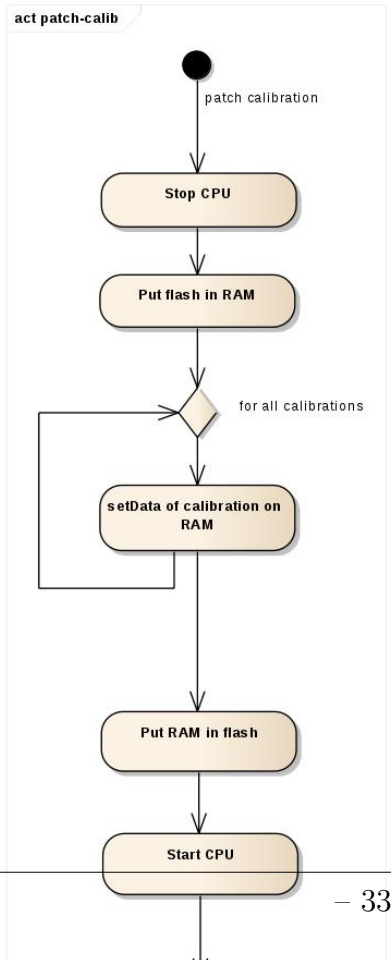
5.2.3 Conception de la solution

Nous avons vu dans la section 5.2.1 que la fonctionnalité était répartie en deux grandes parties, le parser et l'exécution. Ces deux parties peuvent également s'apparenter à deux modules bien distincts de *GreenT* : le client, pour le parser, et le serveur Debugger, pour l'exécution. Dans la suite de ce document, nous feront abstraction de l'utilisation de Thrift pour la communication réseau, celle-ci étant généré comme expliqué section 3.3.3.

Le serveur T32 : Exécution

Comme vu section 4.4.1, le serveur du debugger utilise l'outil Trace32. C'est cet outil qui nous permet d'interagir avec le calculateur embarqué. Cet outil fonctionne principalement en utilisant des scripts batch.
Depuis l'interface de cet outil, il est possible de patcher une calibration comme le montre la figure TODO test.

INSERER ICI UNE CAPTURE D'ÉCRAN



Le script existant Ce script a été pensé pour le mode graphique. L'utilisateur appelle le script, modifie les valeurs des calibrations voulues via l'interface , et clique sur « patch intern flash ». À ce moment là, le patch en flash s'effectue.

Le nouveau script Le nouveau script va effectuer les même actions que le script existant, mais sans aucune attente d'actions utilisateurs. Pour cela, plutôt que d'attendre que l'utilisateur modifies les calibrations, nous aurons une liste calibration à modifier en paramètre.

Afin de modifier une calibration, présente en flash, un certain nombre d'actions sont nécessaires. Ces actions sont montrés dans le diagramme d'activité figure 5.2.

Le service du serveur va ainsi appeler le nouveau script de patch, le code Java concerné sera donc minime. Le prototype de la méthode est la suivante :

```
public void patchCalibs(Map<String, Long> calibs);
```

Le client : Parser & générateur

Afin de gérer le parsing et la génération de notre patch calib, j'ai du créer une nouvelle règle de grammaire comme montré section 5.2.2.

Avec cette nouvelle grammaire, j'ai forcé l'utilisateur à ne modifier des calibrations que si l'ECU est éteint, cette action n'étant possible que dans une `patch section`. Une fois la liste de calibration à patcher parsée, il m'a suffit de générer l'instruction d'appel du Debugger.

```
Map<String, Long> calibrations = new HashMap<String, Long>();
calibrations.put("calib1", 0x0);
calibrations.put("calib2", 0x1);
dbg.patchCalibs(calibrations);
// Stimulation instructions
```

Pour répondre à un besoin de l'équipe cliente, j'ai ajouté une sauvegarde de la valeur des calibrations avant le premier scénario, et une restauration à la fin du dernier scenario. Ceci pour garantir une bonne cohérence, en effet, si l'utilisateur modifie une calibration, celle-ci doit être à la valeur initiale pour les tests suivants. Cette restauration se fait de façon implicite afin de limiter le travail des testeurs, qui pourrait être source d'erreur.

Cette sauvegarde se fait simplement, en début de test, je lis la valeur de chacune des calibrations qui seront modifiées par la suite, à la fin du test, je patch chacune des valeurs de sauvegardes en appelant le service.

5.2.4 Les difficultés rencontrées

Afin de développer cette fonctionnalité, j'ai eu quelques problèmes à résoudre. Nous allons voir ici comment j'ai pu passer outre mes problèmes.

Le batch

Tout d'abord, j'ai du modifier un script batch, dans un langage que je connaissais pas, manipulant un outil que je connaissais pas. Afin de pouvoir effectuer mon nouveau script, j'ai donc lu la documentation de l'outil Trac32 expliquant le fonctionnement de leurs scripts, et la manière de les faire. J'ai étalement étudié le fonctionnement du script existant afin de l'adapater. J'ai eu la chance d'étudier un script qui était clair et assez bien construit. J'ai donc été rapidement opérationnel pour comprendre ce que faisait le script, et ce que j'allais devoir faire.

La robustesse

Un problème plus important à cependant été soulevé assez rapidement. Nous avons choisis d'adapter un script Trace32 afin d'être le plus rapide possible, cependant celui-ci nous à soulevé un certain nombre de questions.

En effet, si nous effectuons un patch, que nous coupons l'allimentation de l'ECU, réallimentons celui-ci, et que nous essayons de nouveau de patcher une calibration, une erreur est levée, des bits d'erreurs mis à un, et l'ECU passe dans un état incohérent.

Après investigation, nous avons pu cerner plusieurs sources éventuels de ce problème : Il se pourrait qu'après la coupure de l'allimentation, celui-ci soit refusé en raison d'état de la RAM ou de la flash qui ne soit pas correct.

Ce problème pourrait être localisé dans plusieurs endroits.

Le script existant Ce script est peu utilisé par l'équipe projet, ainsi, il n'est que peu testé. Il se pourrait que nous nous soyions basé sur un script possédant des bogues.

Le driver flash propriétaire Ce script utilise un driver flash propriétaire, dont nous ne connaissons pas exactement les actions. Il se pourrait que le problème vienne également de lui.

Ce problème semble donc compliqué à résoudre, une des solutions étant de développer notre propre driver flash afin d'avoir un contrôle de bout en bout. Cette solution étant particulièrement compliquée et couteuse, j'ai trouvé une autre solution beaucoup plus simple.
La coupure de l'allimentation de l'ECU provoque un état incohérent de la RAM, cependant, si nous redémarrons manuellement l'outil Trace32 entre deux patch, sans coupure brusque, le problème est résolu. Certainement car le démarrage de celui-ci nettoie certains secteurs. Cette solution n'est pas parfaite, mais a pu être effectuée rapidement et fonctionne correctement. Le jour ou de la robustesse sera particulièrement nécessaire, nous devrons développer notre propre driver.

5.3 Les « tableaux calibrables »

Certaines variables côtés ECU sont des variables de types tableau, l'utilisateur peut actuellement y accéder via la syntaxe `var[indice]`.

Le but de mon développement, était d'améliorer ce système. Actuellement, seul un indice numérique peut être renseigné à cette variable, notre client souhaite pouvoir renseigner un indice constant ou une calibration.

5.3.1 Expression du besoin

L'intérêt de renseigner une calibration comme indice de tableau est d'avoir les mêmes calibrations d'un projet, ou d'une *release*, à l'autre, et de pouvoir réutiliser le *Walkthrough* en ne changeant que les valeurs de ces calibrations.
Un autre intérêt de cette fonctionnalité, est de pouvoir recopier scrupuleusement les spécifications, celles-ci étant renseignée via des calibrations.

Avec cette fonctionnalité, l'utilisateur pourra donc avoir des fichiers *Walkthrough* génériques et réutilisable entre chaque versions d'un même projet. Cela limiterai également le risque d'erreur dûe

à une mauvaise recopie de la spécification. En effet, avant cette fonctionnalité, l'utilisateur devait regarder la spécification, aller voir la valeur de la calibration, et la mettre en dur dans le test. Cela peut provoquer des erreurs, d'une part lors de mauvaise recopie de la valeur, mais également si la valeur change à la version suivante et que l'utilisateur oubli de modifier le *Walkthrough*.

Ce besoin concerne uniquement les variables Debugger, et nullement le HiL. En effet, le debugger accède à des variables présente dans le logiciel, dont des tableaux, il est nécessaire de pouvoir y accéder. Cependant, le HiL lui ne connait pas ce mécanisme, il n'utilise que des adresses afin d'accéder à ces élements de stimulation.

Les cas d'utilisation

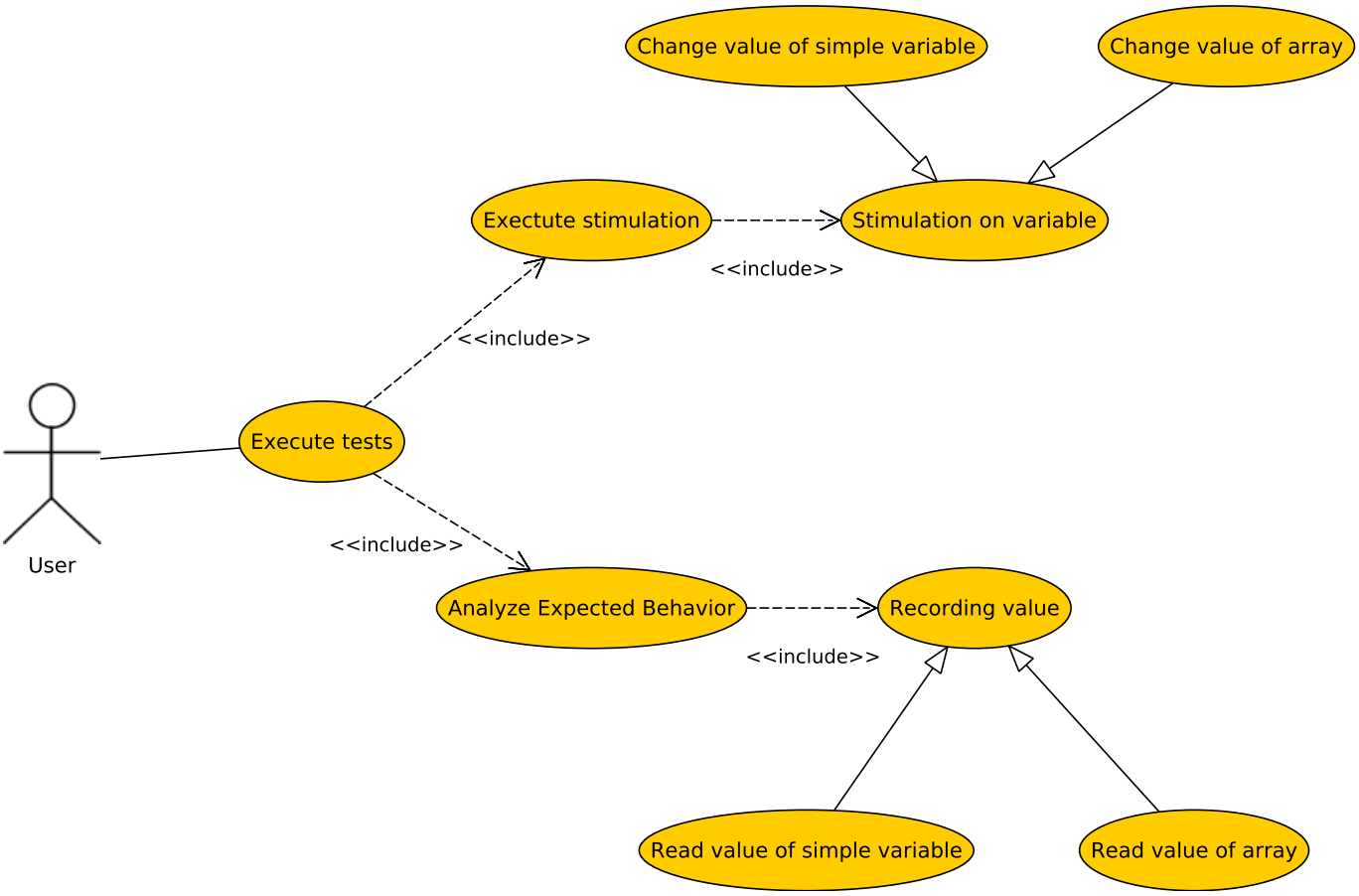


Figure 5.3 – Diagramme de cas d'utilisation des tableaux calibrables

Remarque Cette fonctionnalité possède un cas d'erreurs qui devra être gérés convenablement :
- Si l'utilisateur renseigne un indice en dehors de la taille du tableau, une exception devra être levée
- Si l'utilisateur renseigne autre chose qu'un entier ou une calibration en tant qu'indice, une erreur doit être levée au parsing [a].

a. C'est à la grammaire du langage de ne pas autoriser d'autres indices

Comme nous pouvons le voir dans ce diagramme de cas d'utilisation, le problème est découpé en deux sous-problèmes :
- L'exécution d'une stimulation, modifier une variable tableau sur le debugger, durant un scénario de stimulation.

— L'analyse de l'*Expected Behavior*, c'est-à-dire un type de variable particulier au sein de notre expression logique.

5.3.2 Conception de la solution

L'exécution d'une stimulation

Afin de pouvoir utiliser des tableaux à l'intérieur d'une stimulation j'ai ajouté de nouveaux Alias comme présentés figure . Pour éviter d'eventuels problèmes, je n'ai pas modifié l'existant, simplement ajouté la nouvelle fonctionnalité au sein de ce diagramme de classe, comme vous pouvez le voir avec les classes situés au centre, représenté en orange.

Un alias est un objet qui permet de faire l'abstraction entre le device et l'action que l'on veut faire. Ainsi, ceux-ci sont répartis d'une part par device, et d'autre part en fonction de leur mode d'accès, en lecture ou en écriture. Si un jour un nouveau device est nécessaire, il faudra l'ajouter dans cet arbre d'héritage. Ces alias peuvent effectuer certaines actions en fonction du device concerné, tel que des modification en physique, en hexadécimal ou en eléctrique, et des lecture de valeurs.

Durant ce stage, j'ai ajouté une abstraction permettant de séparer le fonctionnement des Alias « simple » d'un côté, et ceux représentant un tableau. Un tableau pouvant être lui aussi accessible en lecture, ou en écriture. À noter que toutes les actions effectués sur ces alias vont vérifier que l'index demandé existe bel et bien, si ce n'est pas le cas une exception est renvoyée.

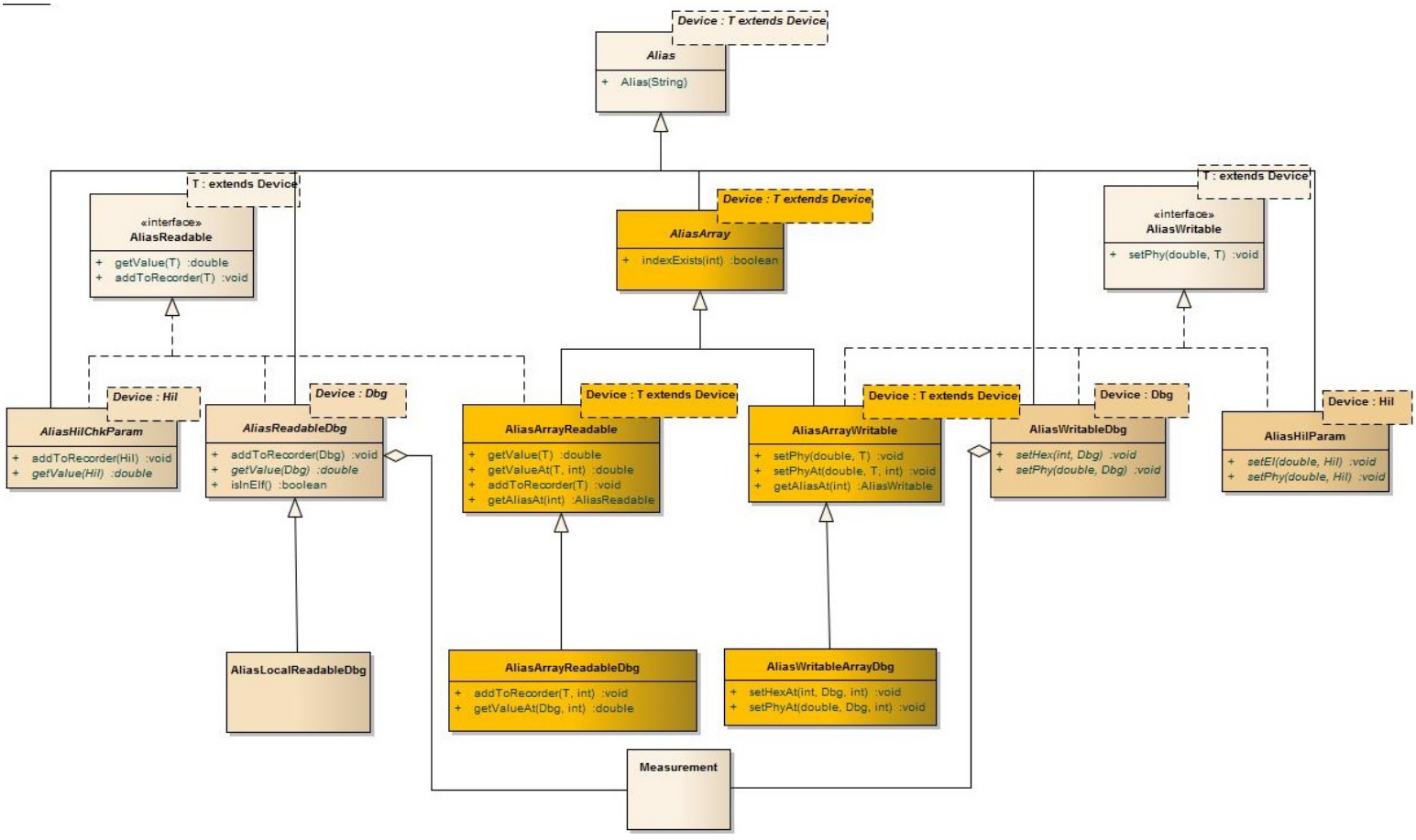


FIGURE 5.4 – Diagramme de classe – Gestion des Alias

Ainsi, les tableaux s'utilisent assez facilement : si nous souhaitons accéder à un élément d'un tableau, il faut appeler les méthodes correspondantes : `setPhyAt`, `getValueAt`, ... Ces méthodes nécessitent un index, et c'est dans cette méthode que je vais effectuer un calcul d'adresse afin d'effectuer mes modifications sur la bonne variable.

L'analyse de l'Expected Behavior

Le stockage de la valeur du tableau Afin d'enregistrer des valeurs durant un scénario, nous utilisons le Debugger, celui-ci génère une trace, que nous transformons ensuite en CSV. Le nom des colonnes sont impérativement le nom de la variable vu par le debugger, c'est-à-dire `variable` ou `tableau[0]`.
L'évaluation de notre *Expected Behavior* se fera en local. Or, nous n'aurons pas la valeur de notre calibration, il sera donc impossible de faire le lien entre `tableau[0]` et `tableau[calib]`, même si `calib` vaut bien 0.
La solution la plus simple pour régler ce problème est d'enregistrer la valeur de la calibration durant le scénario, pour cela, nous pouvons réutiliser la trace, ce qui permet de ne pas redévelopper un mécanisme fonctionnant.

Exemple Si notre *Expected Behavior* est `tab[calib] = var`, et que `calib=0`, alors nous allons enregistrer ceci durant notre scénario :

tab[0]	calib	var
5	0	5

Ce sera au moment de l'évaluation de notre *Expected Behavior* que nous allons faire le lien entre `tab[0]` et `tab[calib]` renseigné par l'utilisateur. Dans ce cas là, notre *Expected Behavior* est vraie.

L'évaluation de l'Expected Behavior Le diagramme de classe présenté figure 5.5 correspond au stockage de notre *Expected Behavior*. Une Expected Behavior contient une liste d'`Evaluable`, correspondant à un arbre logique [2].

La partie que j'ai développé cette année correspond aux deux classes présentes en Orange. Ces deux classes correspondent aux variables présente dans notre arbre d'expression. J'ai simplement ajouté un type de classe permettant de spécifier un tableau calibrable : une variable de tableau calibrable est une variable possédant un indice spécifié par une autre variable.

C'est ensuite via le polymorphisme que j'ai résolu mon problème. Lors de l'analyse d'une *Expected Behavior* sur une trace, avant chaque timestamp, nous allons modifier les valeurs des variables avant de l'évaluer de nouveau. Cette évaluation se fait en faisant le lien entre le nom présent dans la trace, et le nom présent dans le `name` de la variable. Ainsi, mon type de variable particulier retournera en nom, le nom de la variable, ainsi que la valeur de son indice.

Exemple

Timestamp	tab[0]	calib	var
1	5	0	5
2	3	0	3
3	4	0	3

Dans cet exemple, nous allons devoir analyser l'expression à trois reprises : une pour chaque timestamp.

Avant l'évaluation, nous modifions les valeurs des variables de la manière suivante :

2. Les noeuds pouvant être des `AND`, `OR`, `IMPLY`, ...

```
// For Timestamp = 1
eb.getVariable("tab[0]").setValue(5);
eb.getVariable("var").setValue(5);

// For Timestamp = 2
eb.getVariable("tab[0]").setValue(3);
eb.getVariable("var").setValue(3);

// For Timestamp = 3
eb.getVariable("tab[0]").setValue(4);
eb.getVariable("var").setValue(3);
```

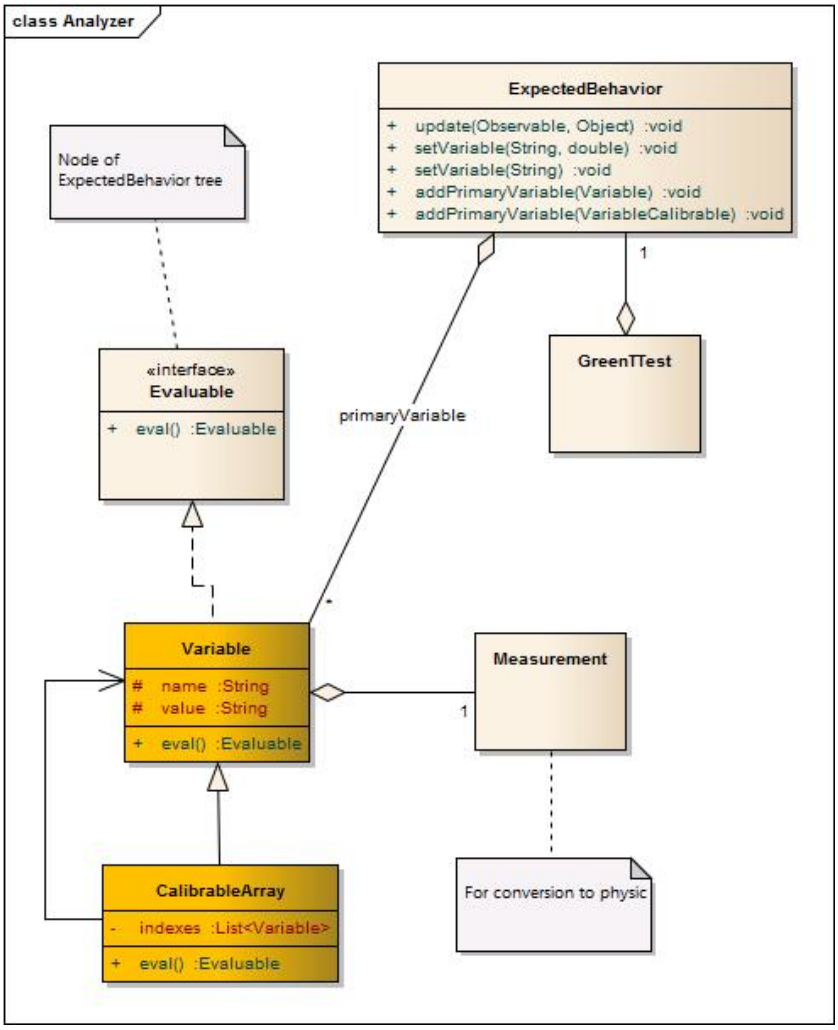


FIGURE 5.5 – Diagramme de classes d'une *Expected Behavior*

5.4 La maintenance

Comme expliqué plus haut, j'ai développé deux fonctionnalités durant ce stage. Cependant, en parallèle de ce développement j'ai également corrigés différents bogues, ou améliorer différentes partie de la plateforme.

Ayant conçu cette plateforme lors de mon stage de fin de Licence, je connais l'intégralité de la plateforme. C'est ainsi que j'ai pu détecter et corriger un certains nombres de problèmes. Ceux-ci ont été identifiés de trois façons différentes :

— Durant mon développement, il m'est arrivé de trouver du code incohérent ou bouchonné
— Lors d'exécutions de la plateforme sur différentes versions du projet client
— En regardant les différents tickets [3] ouverts et devant être résolus.

5.4.1 Corrections

J'ai corrigé quelques problèmes trouvés sur la plateforme, principalement venant du client *GreenT*. Ceci étant dû à notre choix d'architecture : des serveurs les plus légers possibles et un client effectuant le maximum d'actions.

Stockage des erreurs d'exécutions

Le problème En cas d'erreur durant une exécution, si un serveur ne répond plus, si une variable est non trouvée, ... Une exception est levée. À ce moment là, *GreenT* doit attraper l'exception et la stocker en base de données pour pouvoir afficher ensuite le message à tous les tests du bundle, la génération des rapports ne pouvant pas se faire.

La solution Le stockage du message d'erreur n'était pas fait, ainsi que la requête SQL permettant d'obtenir les messages d'erreurs. Ces deux actions ont été corrigés, en lieu et place du rapport de test nous avons maintenant un message d'erreur clair lorsque le test n'a pas pu être exécuté.

Modification des variables Debugger

Le problème Le serveur Trace32 doit pouvoir nous laisser modifier des variables présent dans le logiciel du contrôle moteur. Or, lorsque nous appelions la commande de modification, celle-ci nous renvoyait systématiquement une exception.

La solution Après lecture de la documentation de Trace32, il s'est avéré que le problème venait simplement du serveur qui appliquait une commande syntaxiquement incorrecte. La modification de la commande a corrigé le problème.

Ordre d'exécutions des scénarios

Le problème Un Test peut contenir plusieurs scénarios, ceux-ci nous servent principalement pour le « patch-calib » comme expliqué section 5.2. Or, si nous utilisions plusieurs scénarios, ceux-ci étaient exécutés dans un ordre aléatoire : il n'était pas possible de les exécuter dans un ordre donné.

La solution Le problème avait deux parties : d'une part, l'exécution des scénarios dans un ordre donné, d'autre part, spécifier un ordre à chacun de nos scénarios. En effet, tout d'abord, j'ai nommé les scénarios de sortes qu'ils soient classé par ordre alphabétique. Ensuite, il a fallu spécifier à la plateforme d'exécuter les différents scénarios dans un ordre alphabétique, pour cela il a suffit d'utiliser une collection Java effectuant cette action, la `TreeMap`.

3. Nous avions une liste de «tickets» permettant de connaître les choses à faire sur le client, ou l'un des serveurs. Ces tickets peuvent être une action de maintenance, de *refactoring*, une nouvelle fonctionnalité ou la correction d'un bug. Libre à chacun de mettre à jour ce document

Reset ECU à l'exécution

Le problème Lors de l'exécution de notre plateforme sur la dernière version du projet client, nous avions systématiquement un *reset* ECU. C'est-à-dire que notre ECU s'arrêtait et ne redémarrait pas pour une raison inconnue.

La solution Le problème ne venait pas directement de la plateforme, mais d'une mauvais configuration de notre part. En effet, nous ne spécifions pas les bons fichiers du logiciel, celui-ci étant mal flashé, l'ECU refusait de démarrer.

Absence d'injection

Le problème Lorsque j'essayais de simuler un démarrage du moteur sur la dernière version du logiciel, aucune injection ne se faisait : après le starter, le moteur retournait à zéro tour par minute.

La solution Après s'être renseigné auprès de personnes compétentes, il s'est avéré que cela venait d'un nouveau fichier à flasher dont nous n'avions pas connaissance. Un fichier contenant des calibrations permettant le démarrage du moteur sur table. Ce fichier n'ayant pas été pris en compte durant la conception, j'ai ajouté un nouveau paramètre au fichier de configuration permettant de renseigner des fichiers à flasher additionnels.

5.4.2 Améliorations

Exécution différée

Le besoin Lors de mon développement, j'ai eu souvent des problèmes pour réservés des tables de tests. Celles-ci étant régulièrement prise par les équipes projets.

La solution Afin de ne pas bloquer de tables, et de ne pas retarder notre travail en raison de l'absence de celles-ci, une solution nous est venue : la possibilité de lancer l'outil durant la nuit. En effet, actuellement une exécution dure environ 45 minutes pour 80 tests, après laquelle nous pouvons analyser les rapports et voir les problèmes qui nous sont retournés. Ainsi, j'ai ajouté un nouveau paramètre à l'application permettant de spécifier l'heure à laquelle la génération des `.jar`, la compilation, l'exécution et la génération des rapports va se faire. On peut maintenant lancer une exécution le soir et observer les résultats le lendemain matin.

Passage à Java 8

Le besoin La plateforme fonctionnait sous Java 6. Ainsi, nous allions mettre en production une plateforme déjà obsolète à sa sortie. De plus, les deux versions suivantes de Java propose un certain nombre de fonctionnalités aidant au développement, comme des simplifications d'écriture en Java 7 (*Multi-Catch*, Inférence de type, `switch` sur les strings, ...) ou les lambdas-calculs en Java 8. Enfin, dans un future proche nous aurons besoin d'une interface pour *GreenT*, `JavaFx` serait une bonne solutions, mais celle-ci nécessite Java 8.

La solution Avant de passer à Java 8, il a d'abord fallut vérifier qu'aucune incompatibilité avec les bibliothèques que nous utilisons n'allait apparaître. Ensuite, il était nécessaire de télécharger un compilateur ainsi qu'une JVM, configurer les différents environnements et vérifier qu'une exécution se passait de la même manière que précédemment. Après ce succès, le passage à Java 8 a été concrétisé et permet à notre plateforme de rester moderne !

« Clean-Code »

Le besoin *GreenT* ayant deux ans, et ayant connue quatre développeurs différents, il est parfois nécessaire d'améliorer le code existant ou de le rendre plus lisible.

La solution Lorsqu'en développant mes fonctionnalités ou en corrigeant des bugs je rencontrais du code incompréhensible ou du « code mort », je modifiais celui-ci afin de corriger ces défauts. Ceci permet ainsi de garder un code toujours propre et facile à lire.

Affichage des logs

Le besoin La plateforme effectue beaucoup d'actions, allant du parsing jusqu'à la génération des rapports comme montré chapitre 4. Toutes ces actions doivent être tracés, aussi bien en temps réel, en regardant l'exécution, que plus tard en observant un fichier.

La solution Les logs fonctionnait déjà, en utilisant `log4j`, cependant celui-ci affichait beaucoup trop d'informations en temps réel, et n'affichait pas les informations les plus utiles. Ainsi, j'ai passé en revue la plateforme pour afficher les bonnes actions (Initialisation des bancs, stimulis effectués, affichage des exceptions, ...). Toutes les erreurs sont redirigé vers la sortie des erreurs (`stderr`), et seules les informations les plus importantes sont sur la sortie console (`stdout`). Tous les autres logs, qui peuvent être utile à la compréhension d'un problème et nous aider ne sont accessible que dans nos fichiers de logs. J'ai par ailleurs ajouté un « buffer tournant » permettant aux fichiers de logs de ne jamais dépasser une certaines taille (5Mio), afin de ne pas consommer trop d'espace disque.

6 Bilans

Après ce stage de quatre mois, il est temps de dresser un bilan, du point de vue de Continental afin de voir ce que mon travail leur a apporté, mais également en quoi ce stage a été bénéfique pour ma future carrière professionnelle.

Sommaire

6.1 Bilan pour Continental

Mon travail dans l'équipe de développement aura été intéressant pour l'entreprise, en partie grâce à ma connaissance de la plateforme suite à mon stage de licence. En effet, avoir contribué au projet l'année précédente sur la conception de celui-ci m'a permis de rapidement commencer le travail et de corriger des bugs répartis dans différents modules. De plus, revenir huit mois plus tard sur ce projet m'a permis d'appréhender le logiciel de manière plus globale et j'ai ainsi pu soulever des problèmes que nous n'avions pas vu lors de la conception.

Grâce à mes connaissances de l'architecture j'ai aidé Benjamin Guerin – le troisième membre de l'équipe arrivant sur le projet – j'ai ainsi pu lui donner des explications et des conseils, pendant que lui apportait un regard neuf à l'existant.

Comme présenté chapitre 5, mon travail aura été directement utile à l'équipe de développement et au groupe TAS. En effet, j'ai développé deux nouvelles fonctionnalités attendues par le client mais j'ai également corrigé des bugs. Ces corrections ainsi que les nouvelles fonctionnalités nous permettent maintenant d'exécuter les stimulations ainsi que l'analyse complète sur la dernière version du projet Ford : il est maintenant possible d'avoir les résultats de 79 tests en une heure, contre 51 tests lors de mon arrivée.

Le projet n'est pas terminé, et je n'ai pas pu effectuer toutes les fonctionnalités prévues tel que l'utilisation de GreenT avec d'autres fichiers d'entrées que le *Walkthrough*. Ceci est principalement dû à de mauvaises estimations, notamment en raison de la maintenance demandant plus de temps que prévu. Cependant, je vais continuer ce projet dès septembre en contrat de professionnalisation pendant un an et pourrai ainsi finaliser la plateforme.

6.2 Bilan personnel

Cette expérience en entreprise m'a beaucoup apporté, tout d'abord d'un point de vue technique, j'ai acquis de l'expérience en conception logicielle, grâce à toutes nos réunions où nous réfléchissions à la meilleure approche possible. De plus lors de problèmes, les propositions des autres m'ont permis d'avoir une autre vision du problème et une autre manière de le résoudre !

Contrairement à l'année précédente, j'ai pu travailler directement sur les tables de tests et pu ainsi découvrir des notions d'embarqués et d'automobile, et j'ai pu mieux appréhender le fonctionnement de l'ECU.

Mais j'ai aussi acquis des connaissances humaines avec notamment le travail en équipe, communiquer sur nos avancements, de manière écrite ou orale, et être capable de synthétiser ses propositions de manière claire et concise.

J'ai également eu le plaisir de revenir dans une multinationale, avec des collègues souhaitant toujours transmettre leurs connaissances et leur expérience, notamment dans le monde de l'automobile et de l'embarqué.

Un bilan très positif donc, qui m'a réconforté dans mon projet professionnel : ma continuation en M2 Développement Logiciel, en alternance chez Continental.

Annexes

Ci-après vous trouverez un certain nombre d'annexes qui pourront vous permettre de mieux comprendre l'étendue de mon travail et de la plateforme qui est en cours de développement.

Vous y trouverez ainsi un glossaire, des références, des exemples de rapport, de fichiers générés ainsi qu'une explication plus approfondie des équipements utilisés.

A Acronymes et Glossaire

API Application Programming Interface, ensemble normalisé de classes, de méthodes ou de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels.

Antlr Another Tool for Language Recognition, outil permettant de faciliter l'interprétation d'une chaîne de caractère, celui-ci prend en entrée une grammaire, et génère un arbre syntaxique dans plusieurs langages.

Calibration Valeur stockée en flash pouvant contenir une information permettant de simplifier la configuration véhicule. Une calibration pourrait être le nombre d'injecteurs.

ControlDesk Outil permettant de piloter le HIL, l'interface permet ainsi de modifier des valeurs de l'environnement véhicule, ou de pouvoir les lire graphiquement.

Device Les différents équipements dont pourrait avoir besoin l'utilisateur : Hil, Debugger, ...

ECU Electronic Control Unit, calculateur du contrôle moteur

Excel Logiciel tableur appartenant à la suite de Microsoft Office®. Il est possible de modifier une feuille de calcul depuis un logiciel ou un script, notamment en Java.

Flash La mémoire flash est une mémoire de masse non volatile et réinscriptible. Ainsi les données sont conservées même si l'alimentation est coupée.

Flasher Action d'écrire sur la flash, dans notre cas il s'agit d'écrire ou de mettre à jour le logiciel présent sur la mémoire flash de l'ECU.

Grammaire Formalisme permettant de définir une syntaxe clair et non ambigüe.

HIL Hardware in the loop, permet de simuler un environnement véhicule autour du calculateur du contrôleur moteur : celui-ci réagira comme s'il était embarqué dans une voiture.

JAR Java ARchive est un fichier ZIP utilisé pour distribuer un ensemble de classes Java.

Java Langage de programmation orienté Objet soutenu par Oracle. Les exécutables Java fonctionnent sur une machine virtuelle Java et permettent d'avoir un code qui soit portable peut importe l'hôte.

JSON JavaScript Object Notation est un format de données textuelles, générique, dérivé de la notation du langage JavaScript, il permet de représenter de l'information structurée.

JVM Java Virtual Machine

Logiciel de versionnement Logiciel, tel que *Git*, permettant de maintenir facilement toutes les versions d'un logiciel, mais aussi facilitant le travail collaboratif.

Parsing Processus d'analyser de chaîne de caractère, en supposant que la chaîne respecte un certain formalisme.

Python Langage intérprété de programmation multi-plateforme et multi-paradigme. Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire et d'un système de gestion d'exceptions.

Release Version d'un logiciel qui correspond à un état donné de l'évolution d'un produit logiciel utilisant le versionnage. Ainsi, chez Continental, un projet comporte une multitude de versions différentes.

Apache Thrift Langage de définition d'interface conçu pour la création et la définition de services pour de nombreux langages. Il est ainsi possible de faire communiquer deux problèmes dans deux langages différents : Python et Java dans notre cas.

Trace32 Debugger, permet de debugger un programme embarqué, ceci en permettant de lire la mémoire, mettant des points d'arrêts, ...

UML Unified Modeling Language est un langage de modélisation graphique. Il est utilisé en développement logiciel et en conception orienté Objet afin de représenter facilement un problème et sa solution.

XML Extensible Markup Language est un langage de balisage générique permettant de stocker des données textuelles sous forme d'information structurée.

B Références

Voici les références que j'ai utilié durant ce stage : particulièrement de la documentation, quelques livres dans lesquels j'ai lu les chapitres qui m'intéressaient pour résoudre un certain problème, certains cours enseignés durant mon cursus m'ont été utiles, et enfin des sites web et forums me permettant de résoudre des problèmes spécifiques.

B.1 Documentations en ligne

Documentation Thrift https://thrift.apache.org/docs/

Documentation Antlr http://www.antlr.org/api/

Documentation Freemarker http://freemarker.org/docs/

Documentation Git http://git-scm.com/documentation

Documentation Java http://docs.oracle.com/javase/6/docs/api/

Documentation Python https://docs.python.org/2.6/

B.2 Livres

UML2 par la pratique – Étude de cas et exercices corrigés sixième édition, 2008. Rédigé par Pascal Roques

Design Patterns : Elements of Reusable Object-Oriented Software Second edition, 1999. Rédigé par le Gang of four : Erich Gamma, Richar Helm, Ralph Johnson et John Vlissides.

The definitive Antlr4 reference 2013, Rédigé par Terence Parr

The LaTeX Companion 2nd édition, 2004. Rédigé par Franck Mittelbach, Michel Goossens, Johannes Braams, David carlisle et Chris Rowley

B.3 Cours magistraux

Design pattern 2015, M1 Informatique Développement Logiciel, Jean-Paul Arcangeli

Modélisation Conception Programmation Orienté Objet 2014, M1 Informatique, Ilena Ober

Langages et automates 2013, L3 Informatique, Christine Maurel et Jean-Paul Arcangeli

Construction et réutilisation de composants logiciels 2014, L3 Informatique parcours ISI, Christelle Chaudet

Conception UML 2012, DUT Informatique, Thierry Millan
Qualité logicielle 2012, DUT Informatique, Thierry Millan

B.4 Sites Web et forums

StackOverflow http://stackoverflow.com
Developpez http://www.developpez.com/
Wikipédia http://en.wikipedia.org/wiki/

C Exemple de rapport généré par GreenT

D Exemples de fichiers générés

D.1 Exemple de StimScenario

```
package com.continental.gt.generation.test.stim;

import java.util.List;

import com.continental.gt.test.stim.StimScenario;

import com.continental.gt.devices.Device;
import com.continental.gt.exception.CheckFailedGreenTTexception;

import com.continental.gt.devices.Hil;
import com.continental.gt.test.alias.AliasHilParam;
import com.continental.gt.test.alias.AliasReadable;

import org.apache.thrift.TException;

/**
 * Test of stubbed class generated
 * Generated by GreenT
 */
public class StimScenario_Stub_1 extends StimScenario {

  public StimScenario_Stub_1(){
    super("Anonymous StimScenario_Stub_1");
  }

  public StimScenario_Stub_1(String name) {
    super(name);
  }

  @Override
  public void addAllRequiredAlias() {
    addAliasWritable(Hil.class, new AliasHilParam("HIL_VS"));
    addAliasReadable(Hil.class, new AliasReadable("HIL_VS_OUT"));
    addAliasWritable(Hil.class, new AliasHilParam("HIL_VS"));
    addAliasReadable(Hil.class, new AliasReadable("HIL_VS_OUT"));
    addAliasWritable(Hil.class, new AliasHilParam("HIL_VS"));
    addAliasReadable(Hil.class, new AliasReadable("HIL_VS_OUT"));
    addAliasWritable(Hil.class, new AliasHilParam("HIL_KEY"));
    addAliasWritable(Hil.class, new AliasHilParam("HIL_VB"));
  }
  /*
   * @see com.continental.gt.test.stim.StimScenario#exec()
   * Generated by GreenT.
   */
  @Override
```

```
  public void exec(List<Device> devices) throws CheckFailedGreenTTexception {
    showMsg(".exec() : executing stimulation code of StimScenario_Stub_1 ↩
       class...");
    try {
      double n;
      Thread.sleep(500); // TODO remove me
      Hil hil = (Hil)getDeviceByClass(devices, Hil.class);
      Dbg dbg = (Dbg)getDeviceByClass(devices, Dbg.class);

      ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_VS"))).setPhy(0, hil);

      n = getAliasReadable(Hil.class,"HIL_VS_OUT").getValue(hil);
      if(!(n >= -0.2 && n <= 0.2)) {
        throw new ↩
            CheckFailedGreenTTexception("CHECK(HIL_VS_OUT,0.0,TOLRES(1.0))");
      };
      ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_VS"))).setPhy(0, hil);

      n = getAliasReadable(Hil.class,"HIL_VS_OUT").getValue(hil);
      if(!(n >= 49.5 && n <= 50.5)) {
        throw new ↩
            CheckFailedGreenTTexception("CHECK(HIL_VS_OUT,50.0,TOLPER(1.0))");
      };
      ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_VS"))).setPhy(0, hil);

      n = getAliasReadable(Hil.class,"HIL_VS_OUT").getValue(hil);
      if(!(n >= -0.2 && n <= 0.2)) {
        throw new ↩
            CheckFailedGreenTTexception("CHECK(HIL_VS_OUT,0.0,TOLRES(1.0))");
      };
      dbg.stop();
      ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_KEY"))).setPhy(0, hil);
      ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_VB"))).setPhy(0, hil);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (TException e) {
      e.printStackTrace();
    }

    showMsg("... complete ok!");
  }

}
```

Extrait de code D.1 – Exemple de StimScenario

D.2 Exemple de GreenTTest

```java
package com.continental.gt.generation.test;

import com.continental.gt.test.GreenTTest;
import com.continental.gt.test.report.Severity;
import com.continental.gt.test.alias.AliasReadable;

import org.apache.thrift.TException;

/**
 * Test of stubbed class generated
 * Generated by GreenT
 */
public class GreenTTest_AIRT_Air_uRawTCACDsB2 extends GreenTTest {

  public GreenTTest_AIRT_Air_uRawTCACDsB2() {
    super("Anonymous GreenTTest_AIRT_Air_uRawTCACDsB2");
  }

  public GreenTTest_AIRT_Air_uRawTCACDsB2(String name) {
    super(name);
  }

  @Override
  public void addAllRequiredContextualData() {
    addRecordedVariable(new AliasReadable("Air_uRawTCACDsB2"));
    addRecordedVariable(new AliasReadable("HIL_VB_OUT"));
    addRecordedVariable(new AliasReadable("HIL_KEY_OUT"));
  }

  @Override
  public void createReport() {
    report.setVariableLongName("Sensed value of down stream charged air ↩
       temperature (bank-2)");
    report.setVariableName("Air_uRawTCACDsB2");
    report.setResponsible("D.Matichard (FSD ALTEN09)");
    report.setSeverity(Severity.LOW);
    report.setTestSummary("Test if the interface Air_uRawTCACDsB2 is correctly ↩
       stub to 0");

    report.addComment("");
    report.addComment("");
    report.addComment("");
  }

  @Override
  public void verdict() {
    // TODO Auto-generated method stub

  }
}
```

Extrait de code D.2 – Exemple de GreenTTest

E Liste des codes sources

Table des figures